Thesis for the Degree of Doctor Of Philosophy

Verifying Information Flow Control Libraries

Marco Vassena

Department of Computer Science and Engineering
Chalmers University of Technology

Göteborg, Sweden 2019

Verifying Information Flow Control Libraries
Marco Vassena
Göteborg, Sweden 2019

ISBN 978-91-7597-867-3
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 4548
ISSN 0346-718X

Technical Report 170D
Department of Computer Science and Engineering
Research group: Information Security

Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Abstract

Information Flow Control (IFC) is a principled approach to protecting the confidentiality and integrity of data in software systems. Intuitively, IFC systems associate data with security labels that track and restrict flows of information throughout a program in order to enforce security. Most IFC techniques require developers to use specific programming languages and tools that require substantial efforts to develop or to adopt. To avoid redundant work and lower the threshold for adopting secure languages, IFC has been embedded in general-purpose languages through software libraries that promote *security-by-construction* with their API.

This thesis makes several contributions to state-of-the-art static (**MAC**) and dynamic IFC libraries (**LIO**) in three areas: expressive power, theoretical IFC foundations and protection against covert channels. Firstly, the thesis gives a *functor* algebraic structure to sensitive data, in a way that it can be processed through classic functional programming patterns that do not incur in security checks. Then, it establishes the formal security guarantees of **MAC**, using the standard proof technique of term erasure, enriched with *two-steps erasure*, a novel idea that simplifies reasoning about advanced programming features, such as exceptions, mutable references and concurrency. Secondly, the thesis demonstrates that the lightweight, but coarse-grained, enforcement of dynamic IFC libraries (e.g., **LIO**) can be as precise and permissive as the fine-grained, but heavyweight, approach of fully-fledged IFC languages. Lastly, the thesis contributes to the design of secure runtime systems that protect IFC libraries, and IFC languages as well, against internal- and external-timing covert channels that leak information through certain runtime system resources and features, such as *lazy evaluation* and *parallelism*.

The results of this thesis are supported with extensive machine-checked proof scripts, consisting of 12,000 lines of code developed in the Agda proof assistant.

Keywords: Information-flow control, noninterference, functional programming, Haskell, Agda.

I would first like to thank my advisor Alejandro Russo, for guiding me into academia and nurturing the researcher inside me. On this journey, I could not have wished for a better mentor than you on my side. Your enthusiasm and dedication will always inspire me.

My gratitude goes to my coauthors: Pablo Buiras, Lucas Waye, Joachim Breitner, Deepak Garg, Vineet Rajani, Deian Stefan, Gary Soeller, Peter Amidon, Matthew Chan and John Renner. This thesis would have not been possible without your help and support.

To Deian Stefan, for believing in my skills and giving me the opportunity to do cutting edge research in sunny San Diego. I admire your motivation and endless energy and I am indebted for all your support and encouragement during my stay. Hoping that our collaboration will continue and remain productive as it started, I already anticipate many more long nights!

At UCSD I have met many kind people that helped me feeling at home. I am very greatful to all of you! A special thanks goes to Gary, Tristan, Alex, John, and Andi for introducing me to the Californian lifestyle: amazing Mexican food, awesome concerts, jam sessions on the beach and just lots of fun!

Throughout these years at Chalmers, I have met many talented and passionate people. Even though some have already moved on and some have just joined, I am still glad to have spent some time with all of you. Thank you Wolfgang, Gerardo, Dave, Pablo, Per, Raúl, Elena, Musard, Steven, (the old) Carlo, Danielito, Daniel, Mauricio, Evgeny, Simon, Alexander, Jeff, Carlo, Iulia, Max, Benjamin, Thomas, Sandro, Sólrúln (and more) for all the amazing time spent together. I traveled the world with you (sometimes on lunch trains)!

Thanks to my new PhD sister and brothers: Elisabet, Agustin, and Nachi. Just like in a family, we take care of each other…and I call you when I forget the office keys!

To my dear friends Carlo, Aura, Evgeny, Ludia, Katja and Pier. Dinners, BBQs, board games, movies, trips, hikes, poker nights, "bohemian" discussions…with you I never get bored! Thank you so much for such a great time and, please, *never change*!

A special mention goes to the band that needs no introduction. Musician by vocation and ~~ameatour~~ professional video makers by necessity, spending time with you is always so. much. fun! Thank you Pier, Carlo, Enzo, Grischa, Evgeny! Even though people might not remember the name(s) of our band, I am sure that they will remember our music!

Un grazie di cuore alla mia famiglia! Mamma e papà, grazie per avermi sempre sostenuto e incoraggiato nelle mie scelte e spinto a dare il meglio di me. Non avrei mai raggiunto questo traguardo senza i vostri insegnamenti. A mia sorella Chiara, il tuo affetto e la tua stima valgono tutto per me.

Thank you Katja for being you: never tired, always positive, and with a beautiful smile on your face. Ljubim te!

CONTENTS

Paper IV

Paper V

Paper VI

CHAPTER

ONE

INTRODUCTION

> Technology trust is a good thing,
> but *control* is a better one.
>
> *Stéphane Nappo*
> *Chief Information Security Officer*
> *Société Générale International Banking*

Computer systems have transformed our society. We live in the Information age, where information technology drives the economy and every day billions of people exchange information through an always-growing number of Internet-connected devices. Through their devices (e.g., smartphones, tablets and laptops), users interact with software, often in the form of *apps*, in most of their activities, from work to leisure. Nowadays, news spread around the world within seconds over the Internet, bots trade on the stock market on behalf of humans, and friends stay in touch on social media, just to name a few. While engaging in these activities, users entrust apps with a large amount of information, including private data as well (e.g., credit card number, phone number and GPS location), and, sometimes *unconsciously*, expect the software to keep it confidential. However, software betrays the trust of their users sometimes: data is valuable in a digital society and gets harvested for profit. Many companies gather personal information for running personalized ads, hackers steal private data for criminal financial gain, corporate espionage, and state-sponsored military intelligence, warfare, and mass surveillance. Furthermore, companies have recently started developing smart artificial agents to perform tasks, thought to be beyond computers' capabilities, such as product recommendations, natural language processing, image recognition, etc. The machine learning algorithms used in the development of these agents are notoriously *data-hungry*, thus exacerbating the phenomenon of unauthorized data collection.

In order to protect users' data from these threats, several approaches have been proposed. Currently, the most widespread countermeasures are based

on discretionary *access control*, wherein access to sensitive data is restricted to limit information leakage at the discretion of the user. However, simply granting or denying access to some piece of data is often a too rigid security policy in practice. Sometimes, software legitimately needs access to sensitive data to fulfill its functionality. Furthermore, even when access to sensitive data is justified, such policies are too coarse-grained: once access is granted, there is no control on what the software will do with the data afterwards. The fact that software routinely incorporates untrusted third-party software components aggravates the situation—access to sensitive data gets *implicitly* extended to those components as well. Then, either trust is extended to these components as well—neglecting the security principle of least privilege—or the functionality of these components must be implemented again from scratch, at the expense of software reusability.

Example. Trip planner apps find an optimal means of traveling between two or more locations. Using the Internet connection and the GPS location of a device, these apps can give accurate real-time directions to users, even when they do not know where they are. However, such a handy functionality comes at a cost: security mechanisms that enforce access control require the user to grant access to the GPS location and the Internet connection of his device. The user is then faced with a decision: either he trusts the app and receives accurate directions, or he denies access and finds directions in another way. Granting access has the risk of disclosing his location to unauthorized parties: access control security policies allow a malicious app to locate the user via the GPS of the device and exfiltrate his position through the Internet connection. However, the safe option (denying access) is not only unfortunate, but also seemingly overly precaucious—even an honest trip planner app needs his location and an Internet connection to compute real-time directions—therefore the user will likely grant access in good faith. Furthermore, even if his trust is well-founded, the app might include third-party software components (e.g., external libraries) that could still leak his location.

As the example above shows, security policies based on access control are insufficient to protect data confidentiality. To restore security, researchers have proposed stricter security policies, which allow unrestricted access to data, but limit *where* it may propagate within a system instead. The security mechanisms that enforce these policies are based on *information-flow control*, which confines data even when manipulated by untrusted software. Even though information-flow control techniques have been studied widely, they have yet to see widespread practical use—they require developers to use of specific programming languages and tools that require substantial efforts to develop and to adopt. To reduce these efforts, researchers have embedded information-flow control in software *libraries* of general-purpose languages. These libraries guarantee that programs written using their programming interface (API) are *secure-by-construction*.

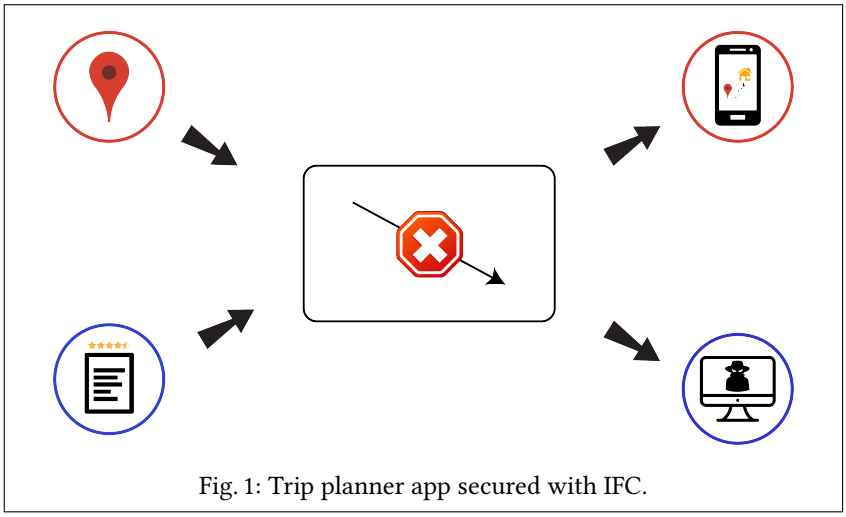


Fig. 1: Trip planner app secured with IFC.

Contributions. This thesis makes several contributions. Firstly, it shows that *state-of-the-art* information-flow control libraries are secure using established formal verification approaches. Then, it demonstrates that the lightweight, but coarse-grained, security enforcement techniques used in these libraries can be as precise and permissive as the fine-grained, but heavyweight, techniques of fully-fledged secure languages. Lastly, this thesis makes contributions to the design of secure runtime systems, which protect information-flow control libraries, and fully-fledged secure languages as well, against leaking information through certain runtime system resources and features.

In the following, we introduce the research area of information-flow control, focusing on relevant techniques and libraries, and conclude with a more detailed overview of the contributions of this thesis.

1 Information-Flow Control

In this thesis, we investigate information-flow control (IFC), a security mechanism that enforces end-to-end data confidentiality and integrity [128]. In contrast to access control, information-flow control does not restrict access to sensitive information, but rather restricts *where* information of different sensitivity levels may propagate within a system. Intuitively, IFC systems associate security levels or *labels* with resources (e.g. program variables, threads, processes, sockets etc.) and use them to track flows of information in the system. A security lattice specifies the flow policies, that is how information is allowed to flow between labeled resources [37]. Then, the IFC mechanism detects when a forbidden flow occurs and takes action to suppress the leakage.

To illustrate the basic working principles of information-flow control, we revisit the trip planner app from the example above. In Figure 1, the box in the center represents the app, which receives inputs on the left and produces outputs on the right. The inputs and outputs are labeled with their security

level, so that the IFC mechanism can track information flows and enforce security. For simplicity, we use a basic security lattice consisting of two labels, i.e., **Public** and **Secret**, and consider flows from a secret input to a public output a security violation. The trip planner app has two inputs, i.e., the user's location, classified as **Secret**, and the review of a restaurant, classified as **Public**, and two output channels, i.e., the display of the device (**Secret**) and an untrusted server controlled by the attacker (**Public**). Using these labels, the IFC mechanism restricts how information propagates from the inputs to the outputs of the app, according to the flow policies of the security lattice. For example, it allows the app to show directions involving the device's location on the display of the device—it is secure to send secret information on a secret channel. On the other hand, the IFC mechanism prevents the app from transmitting the location to the untrusted server: sending secret data on a public channel represents an information leak.

Information-flow control has been applied in various contexts, including operating systems [69], e.g., HiStar [166], Asbestos [39], distributed systems [167], e.g., Laminar [122], Fabric [79, 80], Aeolus [33], cloud computing [11], databases [133, 134], Web frameworks, e.g., SIF [35], Swift [34], Hails [45], Jaqueline [163], JSLINQ [14] LWeb [105], and both imperative and functional programming languages, e.g., Jif [100] and Paragon [28] for Java, JSFlow [51] for Javascript, Jeeves for Python [163] and Scala [164], Flow Caml [137] for Caml, in Spark (a safety critical language subset of Ada) [117] and in many Haskell libraries [2, 9, 30, 75, 76, 123, 124, 131, 144, 145, 149]. In this thesis, we focus on language-based IFC mechanisms, which inspect the code of a computer program to track information flows precisely. Since these techniques enforce security systematically throughout the code, programs are secure-by-construction and do not leak information accidentally, e.g., due to software bugs, or otherwise, e.g., when combined with adversarial code. Most IFC systems enforce a security property called *non-interference* [47], which, intuitively, guarantees that secrets inputs of a program do not affect its public outputs, like in the example above. In practice, the enforcement mechanisms of IFC systems vary considerably. In the following, we give a brief overview of language-based IFC techniques.

1.1 Facets of Information-Flow Control

When. Information-flow control can enforce security *statically* at compile time, *dynamically* at run-time, or both at compile and run-time in a *hybrid* fashion. Static approaches consist of a program analysis, often in the form of a security type system, that rejects possibly leaky programs before execution [1, 28, 82, 99, 123, 138, 157, 158]. Conversely, dynamic techniques monitor program execution and abort a computation, or throw a runtime exception, when it would leak information otherwise [7, 9, 55, 144]. Static approaches are attractive, because secure programs do not incur in any performance overhead at run-time. However, sound analyses must determine the absence of leaks in *all* possible executions of a program. In practice, most analyses are

incomplete and conservatively reject secure programs as well sometimes. Dynamic approaches are more permissive, e.g., with respect to dead code, because they accept or reject a *single* program execution, instead of an entire program, but they might still trigger false alarms. Furthermore, some IFC monitors feature label *introspection* API that allow to inspect and assign security labels at run-time, thus enabling data-dependent security policies, wherein the security level of some piece of data depends on runtime values. Finally, hybrid approaches combine static and dynamic analysis in a single system that mixes static and dynamic types to boost permissiveness, reduce performance overhead, and ease the adoption of security type systems through gradual typing techniques [3, 30, 42, 50, 93, 148]. This thesis makes contributions to both static and dynamic information-flow control (Chapter 2, 3 and 4, 5 respectively).

How. Programs can leak information *explicitly*, for example, by assigning the content of a secret variable to a public variable. Programs can also leak information *implicitly*, via the control-flow structure of a program, for example, if the decision of assigning some value to a public variable depends on secret data. Both dynamic and static IFC mechanisms can detect explicit and implicit flows and take appropriate countermeasures. However, programs can also leak information through *covert channels*, which are features of the system that are not intended to be used for information transfer [72]. For example, programs can signal information through the *termination channel*, i.e., the (non)termination of a computation, the *timing channel*, i.e., the time that a program takes to perform some public action (e.g., to terminate, print or send some data), the *resource exhaustion channel*, i.e., by exhausting some shared finite resource such as memory and disk space, and more (probabilistic channels, power consumption) [128]. Furthermore, the same covert channel can be exploited in different ways, combining shared runtime state and features (e.g., event loops [155], the thread scheduler [17, 125, 126], the garbage collector [107], and lazy evaluation [31]), shared operating system state (e.g., file system locks [65], and events and I/O [61, 78]), and shared hardware (e.g., caches, buses, pipelines and hardware threads [44, 108]).

Usually, leaking information through a covert channel requires carefully crafted code: honest, but buggy, programs are unlikely to leak information that way. However, covert channels pose a real threat for systems that execute untrusted, adversarial code with secret data. In this scenarios, failing to design and implement appropriate countermeasures challenges the applicability of information-flow control. In practice, which covert channels are a concern depends on what attackers can observe of the actual computing system—the number of covert channels and their bandwidth vary depending on that. In order to study the security guarantees of a system in a rigorous way, formal models include an *attacker model*, which specifies what parts of the computing system model can be observed by the adversary. Generally speaking, external attackers can bypass the security mechanisms of most systems—these attackers can make arbitrary precise observations of the system, including execution

time, power consumption and even physical memory. For example, external attackers can break cryptosystems and violate the privacy of users' browsing history through precise timing measurements, via the *external timing* covert channel [24, 41, 49, 67, 160]. In these cases, *mitigation* techniques can, at least, reduce the bandwidth of information leakage [5, 141, 168, 169]. Nevertheless, attackers can also observe timing behavior indirectly, without measuring execution time. For example, in concurrent systems with shared resources, the attacker can learn secret information by affecting the interleaving of threads in a data race, through the *internal timing* covert channel [125, 126, 141, 156]. This vulnerability is more serious, because it does not rely on an external observer, but leaks internally to other threads executing on the IFC system itself. Furthermore, concurrency magnifies the bandwidth of this covert channel to the point where secret information can be leaked efficiently [141].

In Chapter 5 of this thesis, we observe that parallelism *internalizes* many external timing channels and propose countermeasures that close both the internal- and external-timing channels that exploit the runtime system.

What. One of the main distinguishing factor of IFC systems is the *granularity* at which they track information flows. Most IFC operating systems (e.g., [39, 69, 166]) are *coarse-grained*, i.e., they track information flows and enforce flow policies at the granularity of a process or thread. Conversely, most IFC programming languages (e.g., [7, 51, 59, 157, 164, 165]) track information flows in a more *fine-grained* fashion, for example at the granularity of program variables and references. The granularity of the tracking system has important implications for the usage of IFC technology in practice. Firstly, fine-grained systems require considerably more label annotations than coarse-grained systems, where a single label is typically used to protect all the data in scope of a computation. Secondly, coarse-grained systems are often easier to design and implement—they inherently track less information—and can even be embedded in software libraries [2, 9, 30, 75, 76, 123, 124, 131, 144, 145, 149]. However, coarse-grained systems often suffer from the *label creep* problem, wherein the security enforcement is so strict that not even honest programs can accomplish their functionality without triggering false alarms. In contrast, fine-grained systems do not trigger false alarms as often as coarse-grained ones—they are seemingly more flexible, thanks to the fine-grained tracking mechanism.

Contrary to widespread belief and despite all these differences, Rajani and Garg showed that *static* coarse- and fine-grained IFC systems are equally expressive [121]. In Chapter 4, we show a similar result for *dynamic* IFC systems that additionally allow label introspection at run-time.

Proof technique. Modern programming languages rely on abstractions to reduce software complexity. However, enforcing security in expressive IFC languages is hard: many features and abstractions are at odds with security. For example, concurrency simplifies modular programming, but enables leakage through internal timing, locks allow threads to synchronize, but deadlocks can leak information, and exceptions complicate reasoning about implicit flows.

Any programming feature really requires careful reasoning in an IFC system: even a simple **if**-statement can expose users to timing attacks [100, 158]. Even worse, security is *not* a compositional property: if individually secure features are combined in the same language, their interaction might still enable data leakage.

Following a rigorous and principled approach to security, researchers have made significant strides towards establishing sound IFC foundations—many IFC systems now support real-world features and abstractions safely [43, 51, 59, 80, 101, 122, 123, 141, 144, 163, 164]. In this thesis, we reason about security of IFC systems following the same formal approach based on *term erasure* [76], a standard proof technique used to prove noninterference of IFC functional languages [30, 55, 141, 145]. Term erasure is a syntactic rewriting operation that removes all data above the security level of the attacker from a program. Once the erasure function is defined, the proof technique mainly requires to establish a *simulation* property between the execution of a program and the execution of the corresponding *erased* program. Intuitively, the property does not hold if a program leaks secret data: the execution of the erased program would get stuck when using secret data, or differ from the execution of the original program, e.g., by taking a different path, if secret data affects the control flow. Then, term erasure naturally defines an equivalence relation that pairs programs that are indistinguishable to the attacker. The rest of the proof technique simply combines the indistinguishability relation and the simulation property described above to mechanically derive noninterference.

Researchers have studied several variants of noninterference, with strictly increasing levels of security [52]. *Termination-insensitive* noninterference guarantees security of terminating batch-job programs (programs may leak via the termination channel), while *termination-sensitive* noninterference secures even diverging programs. Programs with intermediate behaviors (e.g., inputs and outputs) require a stronger notion of noninterference, based on program progress. In particular, programs that produce the same observable behavior up to (despite) a silent divergent point satisfy *progress-insensitive* (*progress-sensitive*) noninterference. Lastly, *timing-sensitive* noninterference gives the strongest security guarantees: it ensures that the execution time of a program (measured as the number of reduction steps) is independent from secret information [84, 158].

This thesis improves term erasure with *two-steps erasure*, a new idea that simplifies reasoning and security proofs for certain problematic language features. In the thesis, we use our technique systematically and prove *progress-sensitive* noninterference for static IFC concurrent calculi in Chapters 2 and 3, *termination-insensitive* noninterference for fine- and coarse-grained dynamic IFC sequential calculi in Chapter 4 and *timing-sensitive* noninterference for a dynamic IFC parallel runtime system in Chapter 5.

2 Information Flow Control Libraries

Haskell [111] is a strongly-typed lazy functional programming language that plays a privileged role in protecting data confidentiality and integrity: it can enforce information-flow control via *libraries* [2, 9, 30, 75, 76, 123, 124, 131, 144, 145, 149]. Extending existing general-purpose languages with IFC mechanisms (e.g., Jif [100] and Paragon [28] for Java, JSFlow [51] for Javascript, Jeeves for Python [163] and Scala [164], Flow Caml [137] for Caml and Spark [117]) requires major engineering effort, because several components (all the way from the compiler to the runtime system) must be adapted to be security-aware. In contrast, adding IFC capabilities via a library-embedded domain specific language (EDSL) is a much more lightweight approach, which avoids redundant work and lowers the threshold for adopting secure languages [75]. The key feature that enables library-based IFC is Haskell type system, which assigns a distinct type to side-effecful (*impure*) code—only code with this special type can perform I/O. Conversely, the type system guarantees that code that does *not* have that type is instead side-effect-free (*pure*). For this reason, only *impure* code requires specific security measures in Haskell—pure code cannot perform any I/O and thus is inherently secure. To this end, these libraries exercise a strict control over side effects and restrict I/O operations in their API, so that they comply with security policies and do not leak data. Intuitively, these libraries encapsulate I/O code inside a *monad*, a programming abstraction for structuring and combining effectful computations [92], and provide only secure I/O API to developers, which then write *secure-by-construction* code. This thesis contributes to two state-of-the-art IFC Haskell libraries, **MAC** and **LIO**.

2.1 MAC

MAC is a *static* coarse-grained IFC library that brings ideas from Mandatory Access Control [19] into a language-based setting [123]. Intuitively, the library takes the *no write-down* and *no read-up* rules as core security principles, which shape the entire library interface to protect data confidentiality. The library annotates the type of side-effecful computations and resources (e.g., data and channels) with a *fixed* security label and provides primitive operations that secure how they interact. Using those labels, **MAC** embeds appropriate flow-policies in the type signature of every primitive operation via type constraints [159], so that the no write-down and no read-up rules are statically enforced. For example, attempting to send secret data on a public channel (like in the program schematized in Figure 1) would violate the *no write-down* security policy embedded in the type signature of some primitive (e.g., function `send`), and would make the program ill-typed. Interestingly, **MAC** embeds a security type system, consisting of the security lattice and the mandatory access control checks, in Haskell vanilla type system through type classes and type constraints (standard *Haskell 98* type system features [111]) and the *Safe Haskell* language extension [147], which ensures that untrusted code respects type safety and module encapsulation. Different from other static IFC libraries [38, 75, 124, 149],

MAC provides many advanced programming features, including exceptions, mutable data structures and concurrency, and it does so in few lines of Haskell code—the whole library counts only 200 LOC. However, in his functional pearl, Russo does not give any formal security guarantees about **MAC**. Instead, he argues that the library is secure because the API design adheres to the security principles of mandatory access control [123].

This thesis establishes the formal security guarantees of **MAC**. Chapter 2 develops a formally verified model of **MAC** and shows that it satisfies *progress-sensitive* noninterference. Chapter 3 strengthens these security guarantees by designing a countermeasure to the internal timing covert channel that exploits Haskell lazy evaluation.

2.2 LIO

LIO is a *dynamic* coarse-grained *floating-label* IFC library [144]. Like **MAC**, **LIO** associates labels to computations and resources to enforce security, but, in contrast, these labels are created at run-time and typically incorporate dynamic information such as usernames or email addresses. To enforce security, **LIO** programs carry a *mutable* label, called *current label*, which, in turn, is used to permit restricted access to I/O operations. Intuitively, in **LIO** reading sensitive data rises the current label, which *floats* above the labels of all data observed by a computation and limits where the rest of the computation may subsequently write. In particular, write operations are subject to a security check: writing to less sensitive resources represents a security violation that triggers a runtime exception. This simple approach protects against *implicit flows* automatically, but can also lead to the *label creep* problem, wherein the current label rises to a point where the computation cannot perform any useful side-effects. To address the label creep problem, developers can restructure their program or, in the sequential setting, execute problematic code that read and manipulate sensitive data in a separate context through primitive `toLabeled`, which then restores the current label to its previous value afterwards. Furthermore, **LIO** programs feature also a *clearance* label, an upper bound on the current floating label, which enables a form of discretionary access control that reduces opportunities to exploit covert channels. **LIO** seems flexible enough for practical use: untrusted **LIO** apps can deliver extended features in `GitStar.com`, a code-hosting website that enforces robust privacy policies on user data [45].

This thesis contributes to **LIO** in two ways. Firstly, Chapter 4 formally proves that dynamic coarse-grained IFC languages like **LIO** can track information flows as precisely as fine-grained IFC languages and thus can be as permissive. Then, Chapter 5 extends **LIO** with parallelism, which enables simultaneous execution of threads on multicore systems. Crucially, the naive interaction between Haskell vanilla runtime system and parallelism internalizes several external-timing channels that could not be observed before, with devastating consequences for security. This thesis presents $\mathbf{LIO}_{\mathrm{PAR}}$, a novel parallel dynamic IFC runtime system design, which eliminates internal- and

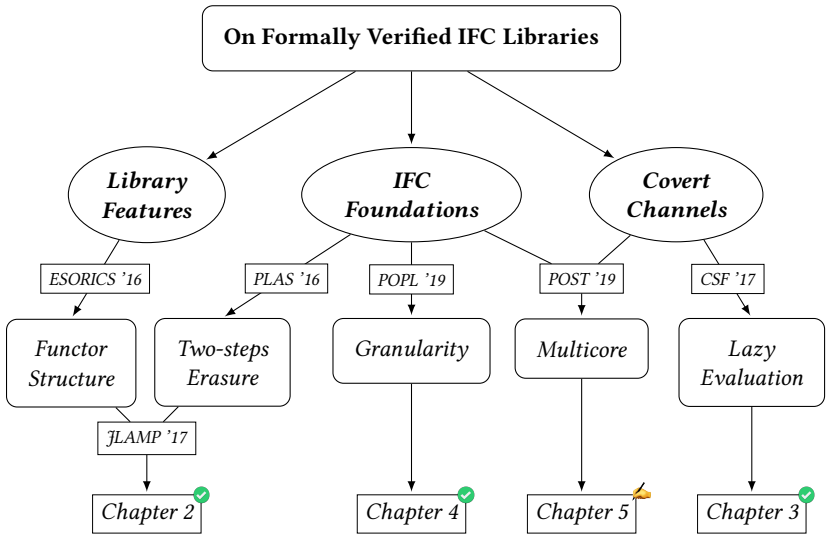


Fig. 2: Overview of this thesis.

external-timing channels that exploit the runtime system in order to restore and strengthen the security guarantees of **LIO** on multicore systems.

3 Contributions

This thesis improves IFC libraries in three areas: library features, IFC foundations and protection against covert channels. Figure 2 summarizes the specific results included in the thesis, relates them to the areas mentioned above, and reports the venues where single research articles have been published or submitted. In the figure, the chapter marked with the hand-writing symbol (✍) features pen-and-paper mathematical security proofs. The check mark (✅) instead identifies the chapters that support their technical results (formal definitions and security proofs) with machine-checked artifacts. These artifacts consist of proof scripts developed in the Agda proof assistant [103], which count more than 12,000 LOC and have been made available online.[1]

Library Features. This thesis enriches labeled data with a *functor* algebraic structure, which enables classic functional programming patterns [89]. The primitive operations of these patterns allow developers to effortlessly process labeled data with *pure* functions, without incurring in the security restrictions of IFC libraries. These primitives promote Haskell idiomatic programming style and enable flexible manipulation of labeled data, and, as a result, make IFC

[1] Links to the machine-checked proof scripts:
- Chapter 2: https://bitbucket.org/MarcoVassena/mac-model
- Chapter 3: https://github.com/marco-vassena/lazy-mac
- Chapter 4: https://hub.docker.com/r/marcovassena/granularity

libraries easier to use and more practical. To ensure that these new operations do not leak data, e.g., when combined with other features of the library, the thesis presents the first comprehensive fully-verified formal model of **MAC** and proves noninterference (Chapter 2).[2]

Foundations. The soundness proof of the model of **MAC** is based on term erasure enriched with *two-steps erasure*, a flexible novel technique useful to reason systematically about the security implications of advanced library features (e.g., exceptions, mutable references and concurrency). Sometimes, simulation via vanilla term erasure falls short for some problematic primitives, e.g., when the sensitivity of some piece of data depends on the surrounding context, and too much or too little information gets erased. Two-steps erasure performs term erasure in two steps. Firstly, it replaces the challenging primitives with special, ad-hoc constructs, then, it performs the desired term erasure during reduction, when sufficient context information is available, through the semantics of these constructs.

Furthermore, this thesis shows that dynamic fine- and coarse-grained IFC languages are equally expressive (Chapter 4). Coarse-grained IFC languages are easier to design and implement—they can even be embedded in software libraries—but are believed to be less permissive than fine-grained languages. We demonstrate that coarse-grained languages can achieve fine-grained precision and permissiveness by presenting a semantics-preserving translation between fine- and coarse-grained languages and use the translation to derive noninterference of each language from that of the other.

Covert Channels. To reduce the costs of implementing secure systems, IFC libraries reuse features of the runtime system of the host language. In general-purpose languages, these features (e.g., schedulers, memory allocators and garbage collectors) are not designed for security and thus make IFC libraries vulnerable to *covert channels*. This thesis makes contributions to the design of secure runtime systems that strengthen the security guarantees of IFC libraries and full-fledged IFC languages as well.

Firstly, the thesis identifies and eliminates external- and internal-timing covert channels that exploit how general-purpose runtime systems manage shared resources (Chapter 5). Intuitively, these runtime systems assign and distribute finite, limited resources (e.g., memory and CPU-time) automatically between threads at different security levels, to guarantee fair access and maximize usage. State-of-the-art libraries like **LIO** and **MAC** ensure that only external attackers can exploit these resource-based covert channels. However, in multi-core systems, threads execute simultaneously on different cores and the covert channel gets *internalized*—threads that execute in parallel are essentially external to one another. In response to these vulnerabilities, the thesis presents the design of $\mathbf{LIO}_{\mathrm{PAR}}$, a new dynamic IFC parallel runtime system that eliminates

[2] Chapter 2 focuses on **MAC** because it lacks formal security guarantees. However, these results extend to other IFC libraries as well.

resource-based timing channels by making resources management hierarchic and explicit at the language level.

Secondly, this thesis addresses the internal-timing covert channel that arises from Haskell *lazy evaluation* (Chapter 3). Interestingly, lazy evaluation combines two features that have opposite security implications: *non-strict evaluation* and *sharing*. Informally, non-strict evaluation guarantees that function arguments are not evaluated until needed inside a function, thus naturally stopping some termination leaks [130]. Instead, sharing is responsible for caching partial results of a program to speed up the rest of the computation. Crucially, sharing is a subtle side-effect in disguise: it is pervasive, but implicit—results of *pure* code are shared as well—and thus eludes the security mechanisms of Haskell IFC libraries. To close this covert channel, the thesis presents a novel unsharing primitive, which duplicates cached results lazily to restrict sharing between threads at different security levels.

The thesis focuses on two state-of-the-art libraries, **MAC** and **LIO**, however the ideas and the techniques presented are general and apply to other libraries and IFC systems as well. In the following, we describe in more detail the publications that form the thesis, which is based upon six papers: one is published in a peer-reviewed journal, four are published individually in the proceedings of peer-reviewed international conferences and workshops and one is currently in submission to a peer-reviewed international conference.

3.1 On Formalizing Information-Flow Control Libraries

The paper presents a full-fledged, computer-verified formal model of the **MAC** library as a simply-typed λ-calculus extended with security primitives and advanced features, such as exceptions, mutable references and concurrency. The main contribution of the paper consists of three insights, which empowers *term erasure* with new proof techniques and simplify reasoning about concurrent systems. The paper describes in detail (i) *two-steps erasure*, a novel proof technique to reason about security in presence of advanced stateful features; (ii) *exception masking*, a novel proof technique that simplifies reasoning about the interaction between exceptions and security primitives; (iii) *scheduler parametric proofs*, the security guarantees are valid for a wide range of deterministic schedulers, that we characterize formally with precise scheduler requirements. As a result, we prove that **MAC** is secure under a round-robin scheduler by simply instantiating our main scheduler-parametric theorem. In addition, the insights of the paper and the extensive proof scripts ($\sim$4,000 LOC), led us to uncover some problems in LIO's proofs and propose changes to repair its noninterference guarantees.

Statement of contributions. This paper was coauthored with Alejandro Russo and published in the proceedings of the 11th Workshop on Programming Languages and Analysis for Security (PLAS), 2016. Marco and Alejandro devised the proof techniques, Marco developed the proof scripts of the model and significantly contributed to the writing of the whole paper.

3.2 Flexible Manipulation of Labeled Values for IFC Libraries

This paper explores the algebraic structure of *labeled data*, i.e., an abstract data type that explicitly tags a piece of data with a label, used in IFC libraries to enforce security policies. These security restrictions are unnecessary when *pure* computations process labeled data: pure code cannot perform any I/O and thus is inherently secure. In this paper, we give a *functor* structure to labeled data, which precisely enables this programming pattern. Furthermore, we study an applicative functor operator, which extends this feature to work on multiple labeled values combined by a multi-parameter function, and a relabel primitive that securely upgrades the label of labeled values as needed when aggregating data with heterogeneous labels. These primitives encourage the functional programming style, typical of the host language, i.e., Haskell, and provides flexibility when manipulating labeled data with side-effect free computations, therefore fostering the secure-by-construction programming model.

Statement of contributions. This paper was coauthored with Pablo Buiras, Lucas Waye, and Alejandro Russo and published in the proceedings of the 21st European Symposium on Research in Computer Security (ESORICS), 2016. Alejandro, Pablo and Lucas conceived the idea of adding a functor structure to labeled values. Marco identified some technical problems with that feature and devised a solution. He developed the proof scripts and was responsible for writing most of the paper.

3.3 MAC, a Verified Static Information-Flow Control Library

This journal article merges and revises the previous two papers, in order to provide a uniform, coherent, comprehensive formal model of **MAC**, to integrate more examples of the features of the library, to fix few technical inaccuracies in the semantics of the calculus, to give a full account of the scheduler-parametric progress-sensitive non-interference theorem, and to simplify its proof. An extended abstract based on these papers was accepted at the 28th Nordic Workshop on Programming Theory (NWPT) in 2016, where it has been selected as one of the best contributions and then invited to a special issue of the Journal of Logical and Algebraic Methods in Programming (JLAMP). The thesis includes this full research article, which subsumes the workshop and the conference papers.

Statement of contributions. This paper was coauthored with Alejandro Russo, Pablo Buiras and Lucas Waye and published in the Journal of Logical and Algebraic Methods in Programming (JLAMP) in December 2017. Marco revised and merged the two papers and was responsible for writing most of the paper.

3.4 Securing Lazy Programs against Information Leakage

Lazy evaluation is a distinctive feature of Haskell, the programming language used to implement many state-of-the-art IFC libraries and tools, including **MAC**. *Sharing*, one of the feature of lazy evaluation, is responsible for caching partial results of a program to speed up the rest of the computation, which,

however enables data leakage via the internal timing covert channel. To close this channel, the paper presents `lazyDup`, a new *unsharing* primitive that lazily restricts sharing from secret threads to public threads, thus disabling any data race between public threads that implicitly depends on secrets via sharing. We formally model sharing with Sestoft's abstract machine, extended with a mutable store, which also exhibits sharing (the first of its kind), and adapt the semantics of the calculus to duplicate *thunks*, when needed for security reasons. We show that the calculus satisfies progress-sensitive non-interference and support our results with machine-checked proof scripts (∼4,000 LOC).

Statement of contributions. This paper was coauthored with Joachim Breitner and Alejandro Russo and published in the proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF), 2017. Joachim conceived the first version of the lazy unsharing primitive and Alejandro suggested to use it to close the sharing-based internal-timing covert channel. Marco adapted the primitive for the operational semantics of Sestoft's abstract machine and for mutable references. He also extended **MAC** with the new primitive, he developed the proof scripts and wrote the technical sections of the paper and the examples as well.

3.5 From Fine- to Coarse-Grained Dynamic IFC and Back

One of the main distinguishing factor of IFC techniques is the granularity with which information flows are tracked. For example, *coarse-grained* IFC operating systems track and enforce information flow at the granularity of a process or thread and *fine-grained* dynamic IFC programming languages track information at the granularity of program variables and references. In these systems, the level of granularity has important practical implications: coarse-grained IFC systems are often easier to design and implement than fine-grained systems, but suffer from the label creep problem, which requires developers to compartmentalize their application in order to avoid false alarms. In contrast, fine-grained systems are seemingly more flexible and do not impose this burden, but require considerably more programmer annotations to track information in a fine-grained fashion. This paper removes the division between fine- and coarse-grained dynamic IFC systems and the belief that they are fundamentally different. In particular, it shows that fine- and coarse-grained dynamic IFC are equally expressive. The paper presents a traditional fine-grained system extended with label introspection primitives, as well as a coarse-grained system, and prove a semantics- and security-preserving translations between them. The results of the paper are supported with machine-checked proofs scripts (∼4,000 LOC).

Statement of contributions. This paper was coauthored with Alejandro Russo, Deepak Garg, Vineet Rajani and Deian Stefan and published in the proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2019. The paper was awarded the "Distinguished Paper" distinction by the program committee of the symposium. This award highlights

papers that the POPL program committee thinks should be read by a broad audience due to their relevance, originality, significance and clarity. At most 10% of the accepted papers can be designated as Distinguished Papers and only 6 papers out of 77 were distinguished in 2019. Deepak and Vineet proved a similar results for *static* fine-grained and coarse-grained IFC systems and contributed to the preliminary parts of this work. Marco and Alejandro devised the proof technique, Marco developed the proof scripts and wrote most of the paper.

3.6 Towards Foundations for Parallel IFC Runtime Systems

This paper presents the foundations for a new dynamic IFC *parallel* runtime system, **LIO**$_{\mathrm{PAR}}$. Most existing IFC systems are vulnerable to external timing attacks because they are built atop vanilla runtime systems that do not account for security—these runtime systems allocate and reclaim shared resources, e.g., CPU-time and memory, *fairly* between threads at different security levels. This paper demonstrates with several proof-of-concept attacks, that extending IFC systems (even concurrent systems such as **LIO**) with parallelism leads to the *internalization* of these attacks. In response to these attacks, the paper proposes **LIO**$_{\mathrm{PAR}}$ a novel parallel runtime system design that safely manages shared resources—both CPU-time and memory—by enforcing *hierarchical* resource allocation and reclamation explicitly at the language-level. **LIO**$_{\mathrm{PAR}}$ is the first parallel language-level dynamic IFC runtime system to address both internal and external timing attacks that abuse the runtime system scheduler, memory allocator and garbage collector. The paper formalizes the design of **LIO**$_{\mathrm{PAR}}$ and proves *timing-sensitive* non-interference, even when exposing clock and heap-statistics APIs.

Statement of contributions. This paper is coauthored with Gary Soeller, Peter Amidon, Matthew Chan, John Renner and Deian Stefan and published in the proceedings of the International Conference on Principles of Security and Trust (POST), 2019. Gary and Deian first conceived the idea of a hierarchical IFC parallel runtime system and Gary developed the code of the attacks that leak secret data externally on a single-core and internally on a multi-core machine. Marco was responsible for formalizing the design of the runtime system, the operational semantics, the security proofs and he significantly contributed to the writing of the whole paper. This work was done during a three-months research visit at University of California, San Diego (UCSD).

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM.
2. Maximilian Algehed and Alejandro Russo. Encoding dcc in haskell. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 77–89, New York, NY, USA, 2017. ACM.
3. A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 137–151, July 2015.
4. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.
5. Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.
6. Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.
7. Jean Bacon, David M. Eyers, Thomas F. J.-M. Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter R. Pietzuch. Information flow control for secure cloud computing. *IEEE Transactions on Network and Service Management*, 11:76–89, 2014.
8. Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld. Jslinq: Building secure applications across tiers. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 307–318, New York, NY, USA, 2016. ACM.
9. Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *Special issue of ACM Transactions on Information and System Security (TISSEC)*, 2009.
10. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
11. Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*. ACM, 2007.
12. Niklas Broberg, Bart Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 217–232, Berlin, Heidelberg, 2013. Springer-Verlag.
13. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.
14. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proceedings of the 18th Nordic Conference on Secure IT Systems - Volume 8208*, NordSec 2013, pages 116–122, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

15. Winnie Cheng, Dan R.K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 139–151, Boston, MA, 2012. USENIX.
16. Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 31–44, October 2007. (Best paper award.).
17. Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.
18. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communication of the ACM*, 20(7):504–513, July 1977.
19. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.
20. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *ACM Symposium on Operating Systems Principles*, SOSP. ACM, 2005.
21. Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.
22. L. Fennell and P. Thiemann. Gradual security typing with references. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 224–239, June 2013.
23. Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium*, pages 531–548, 2016.
24. Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27, 2016.
25. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.
26. J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.
27. Helena Handschuh and Howard M. Heys. A timing attack on RC5. In *Proceedings of the Selected Areas in Cryptography*, SAC '98, pages 306–318, Berlin, Heidelberg, 1999. Springer-Verlag.
28. D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 351–365, July 2015.
29. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing*. ACM, 2014.
30. Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, pages 319–347, 2012.

31. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 11–31, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
32. C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
33. Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4):233–254, 1992.
34. Richard A Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems (TOCS)*, 1(3):256–277, 1983.
35. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman,RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.
36. Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP. ACM, 2007.
37. B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.
38. Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 16–, Washington, DC, USA, 2006. IEEE Computer Society.
39. Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, April 2010.
40. S. Lipner, T. Jaeger, and M. E. Zurko. Lessons from vax/svs for high-assurance vm systems. *IEEE Security Privacy*, 10(6):26–35, Nov 2012.
41. Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.
42. Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.
43. Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328, New York, NY, USA, 2015. ACM.
44. Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Comput. Secur.*, 11(4):615–676, July 2003.
45. Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.
46. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
47. S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 146–160, June 2011.

48. Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.
49. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.
50. Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *USENIX Security Symposium*, pages 1119–1136, 2016.
51. Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.
52. James Parker, Niki Vazou, and Michael Hicks. LWeb: Information flow security for multi-tier web applications. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2019.
53. Mathias V. Pedersen and Aslan Askarov. From trash to treasure: Timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 693–709, 2017.
54. Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.
55. Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):1–255, Jan 2003. `http://www.haskell.org/definition/`.
56. Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. Progress-sensitive security for spark. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, pages 20–37, Berlin, Heidelberg, 2016. Springer-Verlag.
57. Vineet Rajani and Deepak Garg. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *Proc. of the IEEE Computer Security Foundations Symp.*, CSF '18. IEEE Computer Society, 2018.
58. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2009.
59. Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 280–288, New York, NY, USA, 2015. ACM.
60. Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 13–24, New York, NY, USA, 2008. ACM.
61. Alejandro Russo and Andrei Sabelfeld. Security for multithreaded programs under cooperative scheduling. In Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, pages 474–480, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
62. Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *The Journal of Logic and Algebraic Programming*, 78(7):593 – 618, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007).
63. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

64. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, Mar 2001.
65. Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1617–1634, New York, NY, USA, 2018. ACM.
66. Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. Selinq: Tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 25–38, New York, NY, USA, 2014. ACM.
67. David Schultz and Barbara Liskov. Ifdb: Decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 43–56, New York, NY, USA, 2013. ACM.
68. V. Simonet. The Flow Caml system. Software release at http://cristal.inria.fr/ simonet/soft/flowcaml/, 2003.
69. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.
70. Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.
71. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27, 2017.
72. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.
73. David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. *SIGPLAN Not.*, 47(12):137–148, September 2012.
74. Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Transactions on Programming Languages and Systems*, 40(4):16:1–16:55, November 2018.
75. Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 187–202, Washington, DC, USA, 2007. IEEE Computer Society.
76. Pepe Vila and Boris Kopf. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 849–864, Vancouver, BC, 2017. USENIX Association.
77. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*, pages 34–43, June 1998.
78. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

79. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.
80. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.
81. Wing H. Wong. Timing attacks on rsa: Revealing your secrets through the fourth dimension. *XRDS*, 11(3):5–5, May 2005.
82. Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *ACM SIGPLAN Notices*, volume 51, pages 631–647. ACM, 2016.
83. Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.
84. Stephan Arthur Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002. AAI3063751.
85. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.
86. Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.
87. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 563–574, New York, NY, USA, 2011. ACM.
88. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *ACM Conference on Programming Language Design and Implementation*. ACM, 2012.

Paper I, II, III

Based on

On Formalizing Information-Flow Control Libraries,

by Marco Vassena and Alejandro Russo,

11th Workshop on Programming Languages and Analysis for Security;

Flexible Manipulation of Labeled Values for IFC Libraries,

by Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo,

21st European Symposium on Research in Computer Security;

MAC, a Verified Static Information-Flow Control Library,

by Marco Vassena, Alejandro Russo, Pablo Buiras and Lucas Waye,

Journal of Logical and Algebraic Methods in Programming.

CHAPTER

TWO

MAC, A VERIFIED IFC LIBRARY

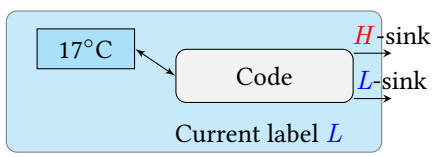
Abstract. The programming language Haskell plays a unique, privileged role in information-flow control (IFC) research: *it is able to enforce information security via libraries*. Many state-of-the-art IFC libraries (e.g., **LIO** and **HLIO**) support a variety of advanced features like mutable data structures, exceptions, and concurrency, whose subtle interaction makes verification of security guarantees challenging. In this work, we focus on **MAC**, a statically-enforced IFC library for Haskell. In **MAC**, like other IFC libraries, computations have a well-established algebraic structure for computations (i.e., monads) responsable to manipulate *labeled values*—values coming from an abstract data type which associates a sensitivity label to a piece of information. In this work, we enrich labeled values with a *functor* structure and provide an *applicative functor* operator which encourages a more functional programming style and simplifies code. Furthermore, we present a full-fledged, mechanically-verified model of **MAC**. Specifically, we show progress-insensitive noninterference for our sequential calculus and pinpoint sufficient requirements on the scheduler to prove progress-sensitive noninterference for our concurrent calculus. For that, we study the security guarantees of **MAC** using *term erasure*, a proof technique that ensures that the same public output should be produced if secrets are erased before or after program execution. As another contribution, we extend term erasure with *two-steps erasure*, a flexible novel technique that greatly simplifies the noninterference proof and helps to prove many advanced features of **MAC**.

1 Introduction

Nowadays, many applications (apps) manipulate users' private data. Such apps *could have been written by anyone* and users who wish to benefit from their functionality are forced to grant them access to their data—something that most users will do without a second thought [91]. Once apps collect users'

information, there are no guarantees about how they handle it, thus leaving room for data theft and data breach by malicious apps. The key to guaranteeing security without sacrificing functionality is not granting or denying access to sensitive data, but rather ensuring that *information flows only into appropriate places*.

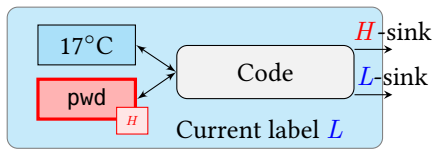
Language-based Information-Flow Control (IFC) [128] is a promising approach to enforcing information security in software. A traditional IFC enforcement scrutinizes how data of different sensitivity levels (e.g., public or private) flows within a program, detects when an unsafe flow of information occurs and takes action to suppress the leakage. To do that, most IFC tools require the design of new languages, compilers, interpreters, or modifications to the runtime, e.g., [28, 99, 116, 122]. Nonetheless, in the functional programming language Haskell, the strict separation between side-effect free and side-effectful code enables lightweight security enforcements. Specifically, it is possible to build a secure programming language atop Haskell, as an embedded domain-specific language that gets distributed and used as a Haskell library [75]. Many of the state-of-the-art Haskell security libraries, namely **LIO** [145], **HLIO** [30], and **MAC** [123], bring ideas from Mandatory Access Control [19] into a language-based setting.[1] These libraries promote a *secure-by-construction* programming model: any program written against their API does not leak secrets. This model is attractive, because it protects not only against benign code that leaks accidentally, e.g., due to a software bug, but also against a malicious program designed to do so. Every computation in such libraries has a *current label* which is used to (i) approximate the sensitivity level of all the data in scope and (ii) restrict subsequent side-effects which might compromise security. IFC uses labels to model the sensitivity of data, which are then organized in a security lattice [37] specifying the allowed flows of information, i.e., $\ell_1 \sqsubseteq \ell_2$ means that data with label ℓ_1 can flow into entities labeled with ℓ_2. Although these libraries are parameterized on the security lattice, for simplicity we focus on the classic two-point lattice with labels H and L to respectively denote secret (high) and public (low) data, and where $H \not\sqsubseteq L$ is the only disallowed flow. The following diagram shows a graphical representation of a public computation in these libraries, i.e., a computation with current label L.



The computation can read or write data in scope, which is considered public (e.g., average temperature of 17°C in the Swedish summer), and it can write

[1] From now on, we simply use the term libraries when referring to **LIO**, **HLIO**, and **MAC**.

to public (L-) or secret (H-) sinks. By contrast, a secret computation, i.e., a computation with current label H, can also read and write data in its scope, which is considered sensitive, but in order to prevent information leaks it can *only* write to sensitive/secret sinks. Structuring computations in this manner ensures that sensitive data does not flow into public entities, a policy known as noninterference [47]. While secure, programming in this model can be overly restrictive for users who want to manipulate differently-labeled values. To address this shortcoming, libraries introduce the notion of a *labeled value* as an abstract data type which protects values with explicit labels, in addition to the current label. The following diagram shows a public computation with access to both public and sensitive pieces of information, such as a password (`pwd`).



Public computations can freely manipulate sensitive labeled values provided that they are treated as black boxes, i.e., they can be stored, retrieved, and passed around as long as its content is not inspected. Libraries **LIO** and **HLIO** even allow public computations to inspect the contents of sensitive labeled values, raising the current label to H to keep track of the fact that a secret is in scope—this variant is known as a *floating-label* system.

Reading sensitive data usually amounts to "tainting" the entire context or ensuring the context is as sensitive as the data being observed. As a result, the system is susceptible to an issue known as *label creep*: reading too many secrets may cause the current label to be so high in the lattice that the computation can no longer perform any useful side effects. To address this problem, libraries provide a primitive which enables public computations to spawn sub-computations that access sensitive labeled values without tainting the parent. In a sequential setting, such sub-computations are implemented by special function calls. In the presence of concurrency, however, they must be executed in a different thread to avoid compromising security through *internal timing* and *termination covert channels* [141].

Practical programs need to manipulate sensitive labeled values by transforming them. It is quite common for these operations to be naturally free of I/O or other side effects, e.g., arithmetical or algebraic operations, especially in applications like image processing, cryptography, or data aggregation for statistical purposes. Writing such functions, known as *pure* functions, is the bread and butter of functional programming style, and is known to improve programmer productivity, encourage code reuse, and reduce the likelihood of bugs [62]. Nevertheless, the programming model involving sub-computations that manipulate secrets forces an imperative style, whereby computations must be structured into separate compartments that must communicate explicitly.

While side-effecting instructions have an underlying algebraic structure, called monad [92], research literature has neglected studying the algebraic structure of labeled values and their consequences for the programming model. To empower programmers with the simpler, functional style, we propose additional operations that allow pure functions to securely manipulate labeled values, specifically by means of a structure similar to *applicative functors* [89]. In particular, this structure is useful in concurrent settings where it is no longer necessary to spawn threads to manipulate sensitive data, thus making the code less imperative (i.e., side-effect free).

Additionally, practical programs often aggregate information from heterogeneous sources. For that, programs need to upgrade labeled values to an upper bound of the labels being involved before data can be combined. In previous incarnations of the libraries, such relabelings require to spawn threads just for that purpose. As before, the reason for that is libraries decoupling every computation which manipulate sensitive data—even those for simply relabeling—so that the internal timing and termination covert channels imposed no threats. In this light, we introduce a primitive to securely relabel labeled values, which can be applied irrespective of the computation's current label and does not require spawning threads.

We provide a mechanized security proof for the security library **MAC** and claim our results also apply to **LIO** and **HLIO**.[2] **MAC** has fewer lines of code and leverages types to enforce confidentiality, thus making it ideal to model its semantics in a dependently-typed language like Agda. The contributions of this paper are:

1. We develop the first exhaustive full-fledged formalization of **MAC**, a state-of-the-art library for Information-Flow Control, in a call-by-need λ-calculus and prove progress-insensitive noninterference (PINI) for the sequential calculus.
2. We enrich the calculus with scheduler-parametric concurrency and prove progress-sensitive noninterference (PSNI) [4] for a wide-range of deterministic schedulers, by formally identifying sufficient requirements on the scheduler to ensure PSNI—a novel aspect if compared with previous work [55, 141]. We leverage on the generality of our result and prove that **MAC** is secure by instantiating our PSNI theorem with a round-robin scheduler, i.e., the scheduler used by GHC's runtime system.
3. We corroborate our results with an extensive mechanized proof developed in the Agda proof assistant that counts more than 4000 lines of code. The mechanization has provided us with stimulating insights and pinpointed problems in proofs of similar works.
4. We improve and simplify the term-erasure proof technique by proposing a novel flexible technique called *two-steps* erasure, which we utilize systematically to prove that many advanced features are secure, especially those that change the security level of other terms and detect exceptions.

[2] The proofs scripts are available at https://bitbucket.org/MarcoVassena/mac-model.

$$
\begin{aligned}
&\textbf{data } \mathit{Labeled}\ \ell\ a = \mathit{Labeled}^{\text{TCB}}\ a \\
&\textbf{data } \mathit{MAC}\ \ell\ a = \mathit{MAC}^{\text{TCB}}\ \{\mathit{run}^{\text{TCB}} :: \mathit{IO}\ a\} \\
&\textbf{instance } \mathit{Monad}\ (\mathit{MAC}\ \ell) \\
&\mathit{label}\ \ \ :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow a \to \mathit{MAC}\ \ell_{\text{L}}\ (\mathit{Labeled}\ \ell_{\text{H}}\ a) \\
&\mathit{unlabel} :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \mathit{Labeled}\ \ell_{\text{L}}\ a \to \mathit{MAC}\ \ell_{\text{H}}\ a
\end{aligned}
$$

Fig. 1: Core API for **MAC**.

5. We introduce a *functor* structure, a relabeling primitive and an *applicative* operator that give flexibility to programmers, by upgrading labeled values and conveniently aggregating heterogeneously labeled data.
6. We have released a prototype of our ideas in the **MAC** library.[3]

Highlights. This work builds on our previous papers "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries" [152] and "On Formalizing Information-Flow Control Libraries" [153], which we have blended and significantly rewritten and corrected in a few technical inaccuracies. We have integrated these works with several examples and shaped them into a uniform, coherent and comprehensive story of this line of work. We summarize the novel contributions of this article as follows:

- Uniform, coherent and comprehensive account of a formal model of **MAC**;
- Integration of examples in the description of the features of the library;
- Fixed several technical inaccuracies in the semantics of the calculus;
- Simplification and full account of the scheduler-parametric PSNI proof.

In the following, we point out the technical differences between this article and the conference version in footnotes.

This paper is organized as follows. Section 2 gives an overview of **MAC**. Section 3 formalizes the core of **MAC** in a simply-typed call-by-need lambda-calculus. Section 4 presents a secure primitive that regulates the interaction between computations at different security levels. Sections 5 and 6 extend the calculus with other advanced practical features, namely exceptions and mutable references. Section 7 proves that the sequential calculus satisfies progress-insensitive noninterference (PINI). Section 8 extends the calculus with concurrency and Section 9 presents functor, applicative, and relabeling operations. Section 10 gives the security guarantee of the concurrent calculus, which satisfies progress-sensitive noninterference (PSNI). Section 11 gives related work and Section 12 concludes.

2 Overview

In **MAC**, each label is represented as an abstract data type. Figure 1 shows the core part of **MAC**'s API. Abstract data type $\mathit{Labeled}\ \ell\ a$ classifies data of type

[3] The **MAC** library is available at https://hackage.haskell.org/package/mac

a with a security label ℓ. For instance, $pwd :: Labeled\ H\ String$ is a sensitive string, while $rating :: Labeled\ L\ Int$ is a public integer. (Symbol :: is used to describe the type of terms in Haskell.) Abstract data type $MAC\ \ell\ a$ denotes a (possibly) side-effectful secure computation which handles information at sensitivity level ℓ and yields a value of type a as a result. A $MAC\ \ell\ a$ computation enjoys a monadic structure, i.e., it is built using the fundamental operations $return :: a \to MAC\ \ell\ a$ and $(\gg\!=) :: MAC\ \ell\ a \to (a \to MAC\ \ell\ b) \to MAC\ \ell\ b$ (read as "bind"). The operation $return\ x$ produces a computation that returns the value denoted by x and produces no side-effects. The function $(\gg\!=)$ is used to *sequence* computations and their corresponding side-effects. Specifically, $m \gg\!= f$ takes a computation m and function f which will be applied to the *result* produced by running m and yields the resulting computation. We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the program $m \gg\!= \lambda x \to return\ (x + 1)$, which adds 1 to the value produced by m, can be written as follows:

$$\begin{array}{l}\textbf{do}\ x \leftarrow m\\ \quad return\ (x+1)\end{array}$$

2.1 Secure Information Flows

Generally speaking, side-effects in a $MAC\ \ell\ a$ computation can be seen as actions which either read or write data. Such actions, however, need to be conceived in a manner that respects the sensitivity of the computations' results as well as the sensitivity of sources and sinks of information modeled as labeled values. The functions *label* and *unlabel* allow $MAC\ \ell\ a$ computations to securely interact with labeled values. To help readers, we indicate the relationship between type variables in their subindexes, i.e., we use ℓ_{L} and ℓ_{H} to attest that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. If a $MAC\ \ell_{\mathrm{L}}$ computation writes data into a sink, the computation should have at most the sensitivity of the sink itself. This restriction, known as *no write-down* [19], respects the sensitivity of the data sink, e.g., the sink never receives data more sensitive than its label. In the case of function *label*, it creates a fresh labeled value, which from the security point of view can be seen as allocating a fresh location in memory and immediately writing a value into it—thus, it applies the no write-down principle. In the type signature of *label*, what appears on the left-hand side of the symbol $\Rightarrow$ are *type constraints*. They represent properties that must be statically fulfilled about the types appearing on the right-hand side of $\Rightarrow$. Type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that when calling $label\ x$ (for some x in scope), the computation creates a labeled value only if ℓ_{L}, i.e. the current label of the computation, is no more confidential than ℓ_{H}, i.e. the sensitivity of the created labeled value. In contrast, a computation $MAC\ \ell_{\mathrm{H}}\ a$ is only allowed to read labeled values at most as sensitive as ℓ_{H}—observe the type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ in the type signature of *unlabel*. This restriction, known as *no read-up* [19], protects the confidentiality degree of the

```
isWeak :: String → IO Bool
p :: IO Bool
p = do
  putStrLn "Choose a password:"
  pwd ← getLine
  return (isWeak pwd)
```

Fig. 2: $isWeak$ can leak the password.

result produced by $MAC\ \ell_H\ a$, i.e. the result might only involve data ℓ_L which is, at most, as sensitive as ℓ_H.

We remark that **MAC** is an embedded domain specific language (EDSL), implemented as a Haskell *library* of around 200 lines of code and programs written in **MAC** are secure-by-construction. What makes it possible to provide strong security guarantees via a library is the fact that Haskell type-system enforces a strict separation between side-effect free code, which is guaranteed *not* to perform side effects, and side-effectful code, where side-effects may occur.[4] Specifically side-effects, i.e., input-output operations, can only occur in monadic computations of type $IO\ a$. Crucially *pure* computations are inherently secure, while IO computations are potentially leaky. In **MAC**, a secure computation of type $MAC\ \ell\ a$ is internally represented as a wrapper around an $IO\ a$ computation, that is used to implement side-effectful features, such as references and concurrency. **MAC** provides security-by-construction because *impure* operations, i.e., those of type IO, can only be constructed using **MAC** label-annotated API, which accepts only those that are statically deemed secure. Function run^{TCB} extracts the underlying $IO\ a$ computation from a secure computation of type $MAC\ \ell\ a$. Thanks to the secure-by-construction design, the IO computation so obtained is secure and can be executed directly, without the need of additional protection mechanism, such as monitors. Note that the function run^{TCB} is part of the Trusted Computing Base (TCB), i.e., it is available only to trusted code. In what follows, we describe an example which illustrates **MAC**'s programming model, particularly the use of $label$, $unlabel$.

Example. The most common use of $label$ is to classify data to be protected. As an example, consider the Haskell program listed in Figure 2, which prompts the user for a password through the terminal and then passes it to a routine to check if the password is listed on dictionaries of commonly used passwords. Observe that the program performs input-output operations: $putStrLn :: String \to IO\ ()$ prints to standard output and $getLine :: IO\ String$ reads from standard input. Clearly the content of variable pwd should be handled with care by $isWeak :: String \to IO\ Bool$. In particular a computation of type $IO\ Bool$ can also perform arbitrary output operations and potentially leak the password.

[4] In the functional programming community, they are also known as *pure* and *impure* code respectively.

```
isWeak :: Labeled H String → MAC L (Labeled H Bool)
p :: IO Bool
p = do putStrLn "Choose a password:"
       pwd ← getLine
       let lpwd = label pwd :: MAC L (Labeled H String)
       Labeled^TCB b ← run^TCB (lpwd >>= isWeak)
       return b
```

Fig. 3: Label H protects the password in $isWeak$.

One way to protect pwd is by writing all password-related operations, like $isWeak$, within **MAC**, where pwd is marked as sensitive data. Adjusting the type of $isWeak$ appropriately, **MAC** prevents intentional or accidental leakage of the password. Several secure designs are possible, depending on how $isWeak$ provides its functionality. For example a secure interface could be $isWeak :: Labeled\ H\ String \rightarrow MAC\ L\ (Labeled\ H\ Bool)$, where the outermost computation ($MAC\ L$) accounts for reading public data, e.g., fetching online dictionaries of common passwords, while the labeled result ($Labeled\ H\ Bool$), protects the sensitivity of this piece of information about the password, namely if it is weak or not. The type $isWeak :: Labeled\ H\ String \rightarrow MAC\ H\ Bool$ is also secure and additionally allows to read from secret channels, e.g., file `/etc/shadow`, to check that the password is not reused. Figure 3 shows the modifications to the code needed to use a secure password strength checker. Observe how $label$ is used to mark pwd as sensitive by wrapping it inside a labeled expression of type $Labeled\ H\ String$. After that, the labeled password is passed to function $isWeak$ by bind ($\gg=$), function run^{TCB} executes the whole computation, whose labeled result is then pattern matched with $Labeled^{TCB}$, exposing the boolean value, that is finally returned.[5]

2.2 Implicit Flows

The interaction between the current label of a computation and the no read-up and no write-down security policies makes implicit flow ill-typed. Consider for instance, the following ill-typed program, which attempts to leak the value of the secret boolean into a public boolean:

```
implicit :: Labeled H Bool → MAC H (Labeled L Bool)
implicit secret = do
  bool ← unlabel secret
  if bool then label True   -- type error: H ⋢ L
          else label False
```

[5] In Figure 3, the code in the IO monad is trusted, hence the use of run^{TCB} and $Labeled^{TCB}$. The function $isWeak$ is not trusted and the password is protected by **MAC** secure API.

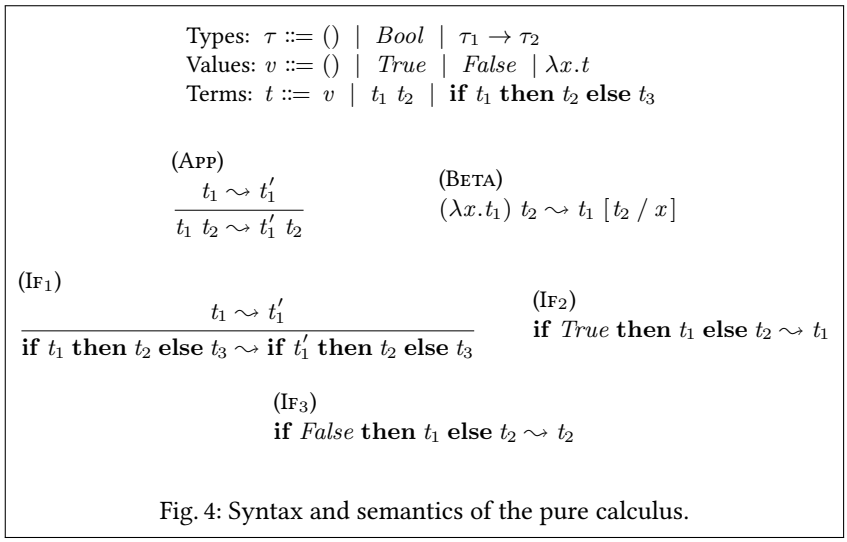


Fig. 4: Syntax and semantics of the pure calculus.

Unlike other IFC system, the code cannot branch on *secret* directly, because it is explicitly labeled (it has type $Labeled\ H\ Bool$ instead of $Bool$). In order to branch on sensitive data, the program needs first to unlabel it, thus incurring in the no read-up restriction that requires the computation to be sensitive as well, that is the program must have type $MAC\ H\ a$ (for some type a). The only primitive that produces labeled data is *label*, which according to the no write-down restriction, prevents a sensitive computation from creating a public labeled value. Then, the program *implicit* from above is ill-typed because it tries to label a piece of data with L in a computation labeled with H, which does not respect the type constraint of *label* ($H \not\sqsubseteq L$).

Features Overview. Modern programming languages provide many abstractions that simplify the development of complex software. In the following, we extend **MAC** with additional primitives that make software development within **MAC** practical, without sacrificing security. The list of programming features securely supported in **MAC** include exception handling (Section 5), references (Section 6), concurrency (Section 8), functors (Section 9) and synchronization primitives (Appendix B).

3 Core Calculus

This section formalizes **MAC** as a simply typed call-by-name λ-calculus extended with unit and boolean values and security primitives.

3.1 Pure Calculus

Figure 4 shows the formal syntax of the pure calculus underlying **MAC**, where meta variables τ, v and t denote respectively types, values, and terms. The typing judgment $\Gamma \vdash t : \tau$ denotes that term t has type τ assuming typing environment Γ. The typing rules of the pure calculus are standard and omitted.

$$
\begin{array}{lll}
\text{Label:} & & \ell \\
\text{Store:} & & \Sigma \\
\text{Types:} & & \tau ::= \cdots \mid \mathit{MAC}\ \ell\ \tau \mid \mathit{Labeled}\ \ell\ \tau \\
\text{Configuration:} & & c ::= \langle \Sigma, t \rangle \\
\text{Values:} & & v ::= \cdots \mid \mathit{return}\ t \mid \mathit{Labeled}\ t \\
\text{Terms:} & & t ::= \cdots \mid t_1 \gg\!= t_2 \mid \mathit{label} \mid \mathit{unlabel}\ t
\end{array}
$$

$$\text{(Lift)}\quad \frac{t \rightsquigarrow t'}{\langle \Sigma, t \rangle \longrightarrow \langle \Sigma, t' \rangle} \qquad \text{(Bind}_1\text{)}\quad \frac{\langle \Sigma, t_1 \rangle \longrightarrow \langle \Sigma', t_1' \rangle}{\langle \Sigma, t_1 \gg\!= t_2 \rangle \longrightarrow \langle \Sigma', t_1' \gg\!= t_2 \rangle}$$

$$\text{(Bind}_2\text{)}\quad \langle \Sigma, \mathit{return}\ t_1 \gg\!= t_2 \rangle \longrightarrow \langle \Sigma, t_2\ t_1 \rangle \qquad \text{(Label)}\quad \langle \Sigma, \mathit{label}\ t \rangle \longrightarrow \langle \Sigma, \mathit{return}\ (\mathit{Labeled}\ t) \rangle$$

$$\text{(Unlabel}_1\text{)}\quad \frac{t \rightsquigarrow t'}{\langle \Sigma, \mathit{unlabel}\ t \rangle \longrightarrow \langle \Sigma, \mathit{unlabel}\ t' \rangle}$$

$$\text{(Unlabel}_2\text{)}\quad \langle \Sigma, \mathit{unlabel}\ (\mathit{Labeled}\ t) \rangle \longrightarrow \langle \Sigma, \mathit{return}\ t \rangle$$

Fig. 5: Core of **MAC**.

The small-step semantics of the the calculus is represented by the relation $t_1 \rightsquigarrow t_2$, which denotes that term t_1 reduces to t_2. Rule [Beta] indicates that the calculus has *call-by-name* semantics, because the argument of a function, evaluated to weak-head normal form by rule [App], is not evaluated upon function application, but rather substituted in the body—we write $t_1\ [x\ /\ t_2]$ for *capture-avoiding substitution*.[6] Rule [If$_1$] evaluates the conditional of an if-then-else expression and rules [If$_2$,If$_3$] take the appropriate branch.

3.2 Impure Calculus

We now extend this standard calculus with the security primitives of **MAC** as shown in Figure 5. Meta variable ℓ ranges over labels, which are assumed to form a lattice $(\mathscr{L}, \sqsubseteq)$. Labels are types in **MAC** despite we place them in a different syntactic category named ℓ—this decision is made merely for clarity of exposition. The new type $\mathit{Labeled}\ \ell\ \tau$ represents a (possibly side-effect free) resource, which annotates a value $t :: \tau$ wrapped in $\mathit{Labeled}$ with the security level ℓ. For example, $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int}$ is a public integer. In the following, we introduce further forms of labeled resources like mutable references (Section 6) and synchronization variables (Appendix B). The actual implementation of **MAC** handles more labeled resources and provides

[6] In the machine-checked proofs all variables are De Bruijn indexes.

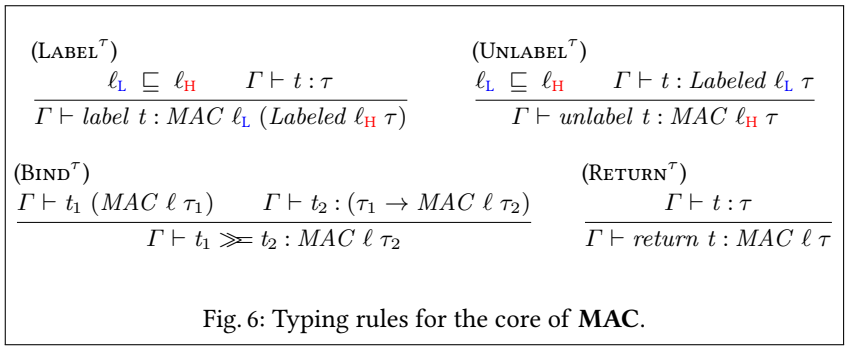


Fig. 6: Typing rules for the core of **MAC**.

a uniform implementation for them [123].[7] The constructor *Labeled* is not available to the user, who can only use *label* and *unlabel* to create and inspect labeled resources, respectively.

A configuration $\langle \Sigma, t \rangle$ consists of a store Σ and a term t describing a computation of type $MAC\ \ell\ \tau$ and represents a secure computation at sensitivity level ℓ, which yields a value of type τ as result. For the moment, we ignore the store in the configuration (explained in Section 6). In order to enforce the security invariants, functions *label* and *unlabel* live in the MAC monad and the typing rules in Figure 6 ensure that the label of the resource is compatible with the security level of the current computation, as explaind in the previous section. We explain the relation between these typing rules and the Haskell API type signatures in Figure 1 as follows. The typing rules in Figure 6 are *type scheme rules*, i.e., there is such a judgment for every label ℓ_L and $\ell_H \in \mathscr{L}$, such that $\ell_L \sqsubseteq \ell_H$, where labels come from either type signatures or explicit type annotations in programs, as we showed in the previous section. The *type constraints* in the API, i.e., what appears before the symbol $\Rightarrow$, is placed as a premise of the corresponding typing rule. We remark that type constrains are built using *type classes*, a well-established feature of Haskell type system, therefore we do not discuss them any further [159]. Besides those primitives, computations are created using the standard monad operations *return* and $\gg\!=$. The primitive *return* lifts a term in the monad at security level ℓ by means of typing rule [RETURN$^\tau$]. Unlike the Dependency Core Calculus (DCC) [1], secure computations at different security levels do not mix in **MAC**: the typing rule [BIND$^\tau$] prevents that from happening—note the same label ℓ is expected both in the types of t_1 and t_2. Just like rules [LABEL$^\tau$,UNLABEL$^\tau$], the typing rules [RETURN$^\tau$,BIND$^\tau$] are type scheme rules, i.e., there is such a rule for each label $\ell \in \mathscr{L}$. For easy exposition, in the following we give the type of **MAC**'s constructs as Haskell APIs.

[7] In our conference version [152,153], we follow the original **MAC** paper [123] and represent all labeled resources using the same labeled data type $Res\ t :: Res\ \ell\ \tau$, where $t :: \tau$ determines the kind of resource. For example $Res\ (Id\ 42) :: Res\ \ell\ (Id\ Int)$ is a term representing a public integer. Here, for clarity of exposition, we use separate data types for each labeled resource. This design choice does not affect our results.

$$
\begin{aligned}
&\mathit{savePwd} :: \mathit{Labeled}\ H\ \mathit{String} \to \mathit{MAC}\ L\ (\mathit{MAC}\ H\ ()) \\
&\mathit{savePwd}\ \mathit{lpwd} = \mathbf{do}\ \mathit{putStrLn}^{\text{MAC}}\ \texttt{"Saving new password"} \\
&\qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{passwd}^{\text{MAC}}\ \mathit{lpwd})
\end{aligned}
$$

Fig. 7: A nested computation that writes at security level L and H.

We explicitly distinguish pure-term evaluation from top-level monadic-term evaluation, similarly to [143]. In the calculus, MAC terms step according to the relation $c_1 \longrightarrow c_2$, which extends the pure evaluation relation $\rightsquigarrow$ via rule [Lift]. The semantics rules in Figure 5 are fairly straightforward and follow the pattern seen in the pure semantics, where some *context-rules*, e.g., [Bind$_1$, Unlabel$_1$] reduce a subterm, and then the interesting rule, e.g. [Bind$_2$, Unlabel$_2$], takes over. For example, rule [Bind$_1$] executes the computation, whose result is then passed to the continuation by means of rule [Bind$_2$]. Rule [Unlabel$_1$] evaluates the argument to a labeled value and rule [Unlabel$_2$] returns its content. Rule [Label] creates a labeled expression by wrapping the argument in *Labeled* and returns it in the security monad. It is worth pointing out that the **MAC** enforces security statically, thus no run-time checks are needed to prevent insecure flows of information in these rules.

4 Addressing Label Creep

Let us continue the password example from the introduction. After checking that the password is strong enough, the program replaces the old password with the new one by updating file `/etc/shadow` with the new hashed password, using primitive $\mathit{passwd}^{\text{MAC}} :: \mathit{Labeled}\ H\ \mathit{String} \to \mathit{MAC}\ H\ ()$—notice that the label of the computation is H, in order to unlabel the password and hash it. (We treat password hashes as confidential data as well, because they could enable *offline* dictionary attacks otherwise.) The program should also inform the user that the password is being saved by printing on the screen a message. Since printing on the screen represents a public write operation, **MAC** assigns type $\mathit{putStrLn}^{\text{MAC}} :: \mathit{String} \to \mathit{MAC}\ L\ ()$ to it. In Figure 7, function $\mathit{savePwd}$ combines $\mathit{passwd}^{\text{MAC}}$ and $\mathit{putStrLn}^{\text{MAC}}$ to save the password and inform the user. Observe that $\mathit{putStrLn}^{\text{MAC}}$ `"Saving new password"` $:: \mathit{MAC}\ L\ ()$ and $\mathit{passwd}^{\text{MAC}}\ \mathit{lpwd} :: \mathit{MAC}\ H\ ()$ belong to different MAC computations, which are *nested* and cannot be merged for security reasons. Intuitively, **MAC** disallows executing those operations in the same computation because otherwise secret data (e.g., the password) could be unlabeled and then leaked on a public channel (e.g., the screen). In particular, primitive bind only allows chaining computations that have the same security label. For example, the program in Figure 8 is rejected as ill-typed, because the first computation is labeled with L and the second computation with H and $L \not\equiv H$:

As a result, programs that handle data and channels with heterogeneous labels necessarily involve nested $MAC\ \ell\ a$ computations in the return type.

```
savePwdBad :: MAC L ()
savePwdBad = do putStrLn^MAC "Saving new password"
                passwd^MAC lpwd   -- type error: L ⋢ H
```

Fig. 8: Ill-typed program ($L \not\sqsubseteq H$).

In this case, the type of $savePwd\ lpwd :: MAC\ L\ (MAC\ H\ ())$ indicates that it is a public computations, which prints on the screen, and that *produces* a sensitive computation $MAC\ H\ Int$, which lastly writes to the sensitive file. Obviously having to nest computations complicates programming with **MAC**.[8] For example, $savePwd\ lpwd$ requires to run the public computation to completion first, and then execute the resulting sensitive computation. We recognize this pattern of returning nested computations as a static version of a problem known in dynamic systems as *label creep* [32, 128]—which occurs when the context gets tainted to the point that no useful operations are allowed anymore.

4.1 Semantics of Join

To alleviate the label creep problem of sequential programs, **MAC** features primitive $join$, which safely integrates more sensitive computations into less sensitive ones.[9] **MAC** assigns the following type signature to primitive $join$:

$$join :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow MAC\ \ell_{\mathrm{H}}\ \tau \rightarrow MAC\ \ell_{\mathrm{L}}\ (Labeled\ \ell_{\mathrm{H}}\ \tau)$$

Intuitively, function $join$ runs the computation of type $MAC\ \ell_{\mathrm{H}}\ \tau$ and wraps the result into a labeled expression to protect its sensitivity. As we will show in Section 7.5, programs written using the monadic API, $label$, $unlabel$, and $join$ satisfy *progress-insensitive noninterference* (PINI), where leaks due to non-termination of programs are ignored. This design decision is similar to that taken by mainstream IFC compilers (e.g., [51,100,137]), where the most effective manner to exploit termination takes exponential time in the size (of bits) of the secret [4].

Figure 9 extends the category of terms with the new term $join\ \ t$. Rule [Join] formalizes the semantics of $join$ using big-step semantics—similar to other work [123, 145], we restrict ourselves to terminating computations. We write $\langle \Sigma, t \rangle \Downarrow \langle \Sigma', v \rangle$ if and only if v is a value and $\langle \Sigma, t \rangle \longrightarrow^* \langle \Sigma', v \rangle$, where relation $\longrightarrow^*$ denotes the reflexive transitive closure of the stepping relation

[8] Remember that Haskell employs lazy evaluation, thus the nested computation is not automatically evaluated, but needs to be explicitly executed. Only trusted code can force the evaluation of MAC computations through run^{TCB}.

[9] The $join$ primitive of **MAC** is unrelated to the monadic primitive $join :: Monad\ m \Rightarrow m\ (m\ a) \rightarrow m\ a$ of the Haskell prelude.

Terms: $t ::= \cdots \mid \mathit{join}\ t$

(Join)
$$\frac{\langle \Sigma, t\rangle \Downarrow \langle \Sigma', \mathit{return}\ t'\rangle}{\langle \Sigma, \mathit{join}\ t\rangle \longrightarrow \langle \Sigma', \mathit{return}\ (\mathit{Labeled}\ t')\rangle}$$

Fig. 9: Calculus with join.

$\longrightarrow$. Rule [Join] executes the secure computation $t \Downarrow \mathit{return}\ t'$ and wraps the result t in $\mathit{Labeled}$ to protect its sensitivity.[10]

Revisited Example. The following program $\mathit{savePwd}'$ simplifies the previous program $\mathit{savePwd}$ in Figure 7 by replacing return with join.

```
savePwd' :: Labeled H String → MAC L ()
savePwd' lpwd = do putStrLn^MAC "Saving new password"
                   join (passwd^MAC lpwd)
                   putStrLn^MAC "Password saved"
```

Compare the two versions of the program. The return type of $\mathit{savePwd}'$ does not involve a nested computation and thus it does not suspend the execution of the sensitive computation $\mathit{passwd}^{\mathsf{MAC}}$, which then occurs directly after the first public print statement.

5 Exception Handling

Exceptions are a standard programming language mechanism that abort the execution of a program when an anomalous condition arises. When a program raises an exception, exception handling primitives allow programmers to detect these conditions and, if recovery is possible, resume execution afterwards. Unfortunately, it is impractical to write real-world programs in **MAC** as it has been presented so far: the library does not feature exception handling primitives. For example, consider again the program $\mathit{savePwd}'$ from above. If primitive $\mathit{passwd}^{\mathsf{MAC}}$ fails due to some IO exception, e.g., file `etc/shadow` has already been opened or does not exist, the whole program crashes. Failing to support exceptions not only makes writing real-world programs impractical, but it may also result in information leakage. Intuitively, exceptions affect the control flow of a program, and an uncaught exception can propagate throughout a program and eventually crash it, potentially suppressing public events. For example, if $\mathit{passwd}^{\mathsf{MAC}}$ throws an exception, the program aborts before printing `"Password saved"` on the screen. Observe that, such behavior constitutes a leak, because the failure comes from a sensitive context, i.e., $\mathit{passwd}^{\mathsf{MAC}}\ \mathit{lpwd}$,

[10] We refrain from using $\mathit{label}\ t'$ because we will soon add exceptions to secure computations.

$$\begin{array}{lll} \text{Types:} & \tau ::= \cdots & \mid \chi \\ \text{Values:} & v ::= \cdots & \mid \xi \mid \mathit{throw}\ t \\ \text{Terms:} & t ::= \cdots & \mid \mathit{catch}\ t_1\ t_2 \end{array}$$

(BIND$_\chi$)
$$\langle \Sigma, \mathit{throw}\ t_1 \gg\!= t_2 \rangle \longrightarrow \langle \Sigma, \mathit{throw}\ t_1 \rangle$$

(CATCH$_1$)
$$\frac{\langle \Sigma, t_1 \rangle \longrightarrow \langle \Sigma', t_1' \rangle}{\langle \Sigma, \mathit{catch}\ t_1\ t_2 \rangle \longrightarrow \langle \Sigma', \mathit{catch}\ t_1'\ t_2 \rangle}$$

(CATCH$_2$)
$$\langle \Sigma, \mathit{catch}\ (\mathit{return}\ t_1)\ t_2 \rangle \longrightarrow \langle \Sigma, \mathit{return}\ t_1 \rangle$$

(CATCH$_3$)
$$\langle \Sigma, \mathit{catch}\ (\mathit{throw}\ t_1)\ t_2 \rangle \longrightarrow \langle \Sigma, t_2\ t_1 \rangle$$

Fig. 10: Exception handling primitives.

and therefore can depend on the value of the secret password. To remedy this situation, **MAC** features the following exception handling primitives, where χ represents the exception type:

$$\begin{array}{l} \mathit{throw} :: \chi \to \mathit{MAC}\ \ell\ \tau \\ \mathit{catch} :: \mathit{MAC}\ \ell\ \tau \to (\chi \to \mathit{MAC}\ \ell\ \tau) \to \mathit{MAC}\ \ell\ \tau \end{array}$$

Primitive *throw* simply raises an exception and aborts the current computation. Then, the exception propagates until it reaches the first surrounding *catch* block, which passes it to the exception handler in order to recover from the failure. Section 5.1 formalizes the semantics of these primitives and Section 5.2 discusses some subtleties in the interaction between exceptions and *join*.

5.1 Calculus

Figure 10 adds the exception type χ and extends the category of terms with the exception value ξ, and exception handling primitives $\mathit{throw}\ t$, and $\mathit{catch}\ t_1\ t_2$.[11] Rule [BIND$_\chi$] aborts a program whenever the first part of the computation raises an exception ($\mathit{throw}\ t$), which simply propagates. Then, term $\mathit{catch}\ t_1\ t_2$ executes the computation t_1 via rule [CATCH$_1$], and returns the result via rule [CATCH$_2$], if the computation succeeds , or executes the exception handler t_2 via rule [CATCH$_3$], if the computation fails with an exception.

5.2 Exceptions and Join

The interplay between exceptions and *join* is delicate and security might be at stake if these two features were combined naively [59,143]. The type signatures

[11] For simplicity, we consider a single exception value $\xi :: \chi$.

Values: $v ::= \cdots \mid \mathit{Labeled}_\chi\ t$

$$(\textsc{Join}_\chi)\quad \frac{\langle \Sigma, t\rangle \Downarrow \langle \Sigma', \mathit{throw}\ t'\rangle}{\langle \Sigma, \mathit{join}\ t\rangle \longrightarrow \langle \Sigma', \mathit{return}\ (\mathit{Labeled}_\chi\ t')\rangle} \qquad (\textsc{Unlabel}_\chi)\quad \mathit{unlabel}\ (\mathit{Labeled}_\chi\ t) \rightsquigarrow \mathit{throw}\ t$$

Fig. 11: Secure interaction between $join$ and exceptions.

of the exception handling primitives show that exceptions can be thrown in any context, but can only be caught in a computation at the same security level for security reasons. Notice that it would insecure to propagate *sensitive exceptions*, i.e., exceptions raised in a sensitive MAC computation, to public computations. Intuitively, sensitive exceptions could affect the control flow of public computations and may leak through implicit flows, e.g., by suppressing an observable event.[12] In our calculus, $join$ is the only primitive that combines computations with different labels and thus is potentially vulnerable to this attack. In order to close leaks via exceptions, **MAC** modifies the semantics of $join$ to *mask* exceptions, preventing them to propagate to less sensitive computations, similar to previous work [59, 143].

To implement this countermeasure, Figure 11 adds a new internal constructor $\mathit{Labeled}_\chi\ t$ that masks an exception $t :: \chi$ inside a labeled value of type $\mathit{Labeled}\ \ell\ \tau$, for some label ℓ and type τ. Then, rule [$\textsc{Join}_\chi$] masks the exception thrown in a nested computation using the new constructor, so that the exception is not propagated further, but rather hidden inside a labeled value. As a result, primitive $join$ always returns a labeled value, regardless of whether the nested computation has succeeded or failed. Since labeled values are opaque, the only way to detect if an exception was raised is to use primitive $unlabel$, which rethrows it via rule [$\textsc{Unlabel}_\chi$]. Since $unlabel$ is subject to the *no read-up* rule, sensitive exceptions remain unobservable from less sensitive computations and only sensitive computations can recover from them. As a result, the program $savePwd'$ from above prints `"Password Saved"` even though $passwd^{\text{MAC}}$ might have actually failed: it would be insecure to do otherwise! The only way to recover from a failure of $passwd^{\text{MAC}}$ without compromising security requires adding an appropriate exception handler, e.g, $\mathit{catch}\ (passwd^{\text{MAC}}\ pwd)\ handler$, and then lifting the whole computation with $join$.

6 References

Mutable references are an imperative feature often needed to boost the performance of programs. Unsurprisingly, references represent yet another communication channel that may lead to information leakage, therefore care is needed when adding this feature to **MAC**. To track information flows and en-

[12] We refer interested readers to [123] for further details about this attack.

```
fetchDict^MAC :: String → MAC L [String]
fetchDict^MAC lang = readFile^TCB ("usr/share/dict-" ++ lang)
fetchCacheDict :: Ref L (Map String [String]) → String
                  → MAC L [String]
fetchCacheDict r lang = do
  dicts ← read r
  case lookup lang dicts of
    Just dict → return dict
    Nothing → do
      dict ← fetchDict^MAC lang
      write (insert dict dicts) r
      return dict
```

Fig. 12: *fetchCacheDict* is a cached version of $fetchDict^{\text{MAC}}$.

force security, **MAC** assigns a fixed security label to references. For example, a reference labeled with ℓ stores data at security level ℓ. Then, **MAC** provides the following monadic API to create, read, and write references securely.[13]

```
data Ref ℓ τ
new :: ℓ_L ⊑ ℓ_H ⇒ τ → MAC ℓ_L (Ref ℓ_H τ)
read :: ℓ_L ⊑ ℓ_H ⇒ Ref ℓ_L τ → MAC ℓ_H τ
write :: ℓ_L ⊑ ℓ_H ⇒ τ → Ref ℓ_H τ → MAC ℓ_L ()
```

The type signature of these operations constraints the labels of the computation and the reference according to the *no write-down* and *no read-up* rules, like those of *label* and *unlabel* in Figure 1. In particular, with these API, computations cannot create or write to less sensitive references and, conversely, read from more sensitive references. The following example illustrates the use of references in **MAC**.

Example. Consider extending the program *isWeak* to reject passwords that are vulnerable to dictionary attacks. To do that, the function $fetchDict^{\text{MAC}}$ in Figure 12 reads a list of words from a dictionary available in the system. Since the content of a dictionary represents public information, **MAC** assigns security level L to $fetchDict^{\text{MAC}}$. The argument of the function specifies the desired language, so that, for example, $fetchDict^{\text{MAC}}$ `"en"` fetches English words from the dictionary contained in the file `"usr/share/dict-en"`. To provide even more robust results, function *isWeak* could test a password against multiple dictionaries and thus call $fetchDict^{\text{MAC}}$ many times. However, dictionaries rarely change: it would be pointless to read the same dictionary multiple times.

[13] **MAC** implements labeled references and their operations as a simple wrapper around standard Haskell references (*IORef*) and their corresponding primitives.

$$
\begin{array}{lll}
\text{Stores:} & \Sigma \in (\ell : \mathit{Label}) \to \mathit{Memory}\ \ell & \\
\text{Memory } \ell\text{:} & t_s ::= [\,] \mid t : t_s & \\
\text{Addresses:} & n \in \mathbb{N} & \\
\text{Types:} & \tau ::= \cdots \mid \mathit{Ref}\ \ell\ \tau & \\
\text{Values:} & v ::= \cdots \mid \mathit{Ref}\ n & \\
\text{Terms:} & t ::= \cdots \mid \mathit{new}\ t \mid \mathit{read}\ t \mid \mathit{write}\ t_1\ t_2 &
\end{array}
$$

(New)
$$\frac{|\Sigma(\ell)| = n}{\langle \Sigma, \mathit{new}\ t\rangle \longrightarrow \langle \Sigma(\ell)[n \mapsto t], \mathit{return}\ (\mathit{Ref}\ n)\rangle}$$

(Write$_1$)
$$\frac{t_1 \rightsquigarrow t_1'}{\langle \Sigma, \mathit{write}\ t_1\ t_2\rangle \longrightarrow \langle \Sigma, \mathit{write}\ t_1'\ t_2\rangle}$$

(Write$_2$)
$$\langle \Sigma, \mathit{write}\ (\mathit{Ref}\ n)\ t\rangle \longrightarrow \langle \Sigma(\ell)[n \mapsto t], \mathit{return}\ ()\rangle$$

(Read$_1$)
$$\frac{t \rightsquigarrow t'}{\langle \Sigma, \mathit{read}\ t\rangle \longrightarrow \langle \Sigma, \mathit{read}\ t'\rangle}$$

(Read$_2$)
$$\langle \Sigma, \mathit{read}\ (\mathit{Ref}\ n)\rangle \longrightarrow \langle \Sigma, \mathit{return}\ \Sigma(\ell)[n]\rangle$$

Fig. 13: **MAC** with references.

To avoid the overhead, references enable a simple caching mechanism. In particular, function $\mathit{fetchCacheDict}$ represents a memoized version of function $\mathit{fetchDict}^{\text{MAC}}$, which takes a public reference to a table of cached dictionaries as an extra argument. When the language *lang* dictionary is needed, the function reads the cached table from the reference ($\mathit{dicts} \leftarrow \mathit{read}\ r$) and checks if it has already been read before ($\mathit{lookup}\ \mathit{lang}\ \mathit{dicts}$). If the table contains the dictionary ($\mathit{Just}\ \mathit{dict}$), then the function simply returns it without performing any IO operation. Otherwise, it reads it from file ($\mathit{dict} \leftarrow \mathit{fetchDict}^{\text{MAC}}\ \mathit{lang}$), caches the result, ($\mathit{write}\ (\mathit{insert}\ \mathit{dict}\ \mathit{dicts})\ r$), and then returns it.

6.1 Semantics

Figure 13 adds mutable references and a memory store to the calculus. Memory is compartmentalized into isolated labeled segments, one for each label of the lattice, and accessed exclusively through the store Σ.[14] For example, memory $\Sigma(\ell)$ belongs to the category $\mathit{Memory}\ \ell$ and contains terms at security level ℓ. For memories, we use the standard list interface: $[\,]$ represents the empty memory, $t : t_s$ denotes prepending term t to memory t_s, and notation $t_s[n]$

[14] A split memory model simplifies the proofs because allocation in one segment does not affect allocation in another. We argue why this model is reasonable and discuss alternatives in Section 7.

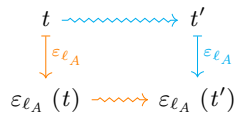


Fig. 14: Single-step simulation.

extracts the nth-element from memory t_s. We write $\Sigma(\ell)[n]$ to retrieve the nth-cell in the ℓ-memory and $\Sigma(\ell)[n \mapsto t]$ for store obtained by updating the n-th cell of memory $\Sigma(\ell)$ with term t. A reference value is written $Ref\ \ n :: Ref\ \ell\ \tau$ where n is an address, pointing to the n-th cell of the ℓ-memory, which contains a term of type τ.[15]

Dynamics. Secure computations create, write and read references using primitives new, $write$ and $read$, respectively. Rule [New] extends the ℓ-labeled memory with the new term and returns a reference to it.[16] The notation $|t_s|$ denotes the length of a list and is used to compute the address of a new reference (memories are zero-indexed). Rule [Write$_1$] evaluates the reference and rule [Write$_2$] overwrites the content of the memory cell pointed by the reference and returns unit. Similarly, [Read$_1$] evaluates the reference and finally rule [Read$_2$] retrieves the corresponding term from the store.

7 Soundness

This section formally presents the security guarantees of the sequential calculus. Section 7.1 gives an overview of the proof technique (*term erasure*), Section 7.2 describes *two-steps erasure*, a novel technique that overcomes some shortcomings of vanilla *term erasure*, Section 7.3 defines the erasure function and Section 7.5 concludes with the *progress-insensitive noninterference* theorem (PINI).

7.1 Term Erasure

Term erasure is a proof technique to prove noninterference in functional programs. It was firstly introduced by Li and Zdancewic [76] and then used in a subsequent series of work on information-flow libraries [55, 124, 141, 143, 145]. The technique relies on an erasure function, written ε_{ℓ_A}, which rewrites data above the attacker's security level ℓ_A to the special syntax node $\bullet$. Once ε_{ℓ_A} is

[15] Even though **MAC** implements labeled reference as a wrapper around Haskell *IORef*, we model references as a simple index into a labeled memory. This design choice does not affect our results.

[16] The labels used in the rules of memory operations (e.g., new, $read$, $write$), come from their typing rules. In our machine-checked proof scripts, terms are annotated with these labels, e.g., $Ref^H\ 0$ denotes that $Ref\ 0$ has type $Ref\ H\ \tau$, which are use in the evaluation rules, e.g., to select the appropriate memory, in which to read a reference.

$$\varepsilon_{\ell_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau) = \begin{cases} \mathit{Labeled}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{Labeled}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{label}\ t :: \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} \mathit{label}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{label}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

Fig. 15: Term erasure for labeled values.

defined, the core of the proof technique consists of proving an essential relationship between the erasure function and the stepping relation. The diagram in Figure 14 highlights this intuition. The diagram shows that erasing sensitive data from a term t and then taking a step (orange path) leads to the same term obtained by firstly taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. Intuitively, if term t leaks sensitive data above ℓ_A, then performing erasure first and then taking a step would not give the same term obtained from first taking a step and then performing erasure—the sensitive data leaked into t' would remain in $\varepsilon_{\ell_A}(t')$. From now on, we refer to this relationship as the *single-step simulation* between regular and erased terms.

7.2 Two Steps Erasure

For certain primitives of **MAC**, the simulation property described above is too strict and does not hold for any definition of the erasure function. Even though these primitives (e.g., new, $write$, $join$) are arguably secure, the erasure function always removes too much or too little information and breaks the commutativity of the simulation diagram. To formally prove security, we have then devised a technique called *two-steps erasure*, which performs erasure in two steps—a novel approach if compared with previous work [145]. Rather than being a pure syntactic procedure, erasure introduces special constructs, which perform erasure through additional evaluation rules. As a result, erasure occurs in two stages along the orange path in Figure 14. In particular, erasure rewrites a problematic primitive to an ad hoc construct, along the vertical solid arrow, and then removes sensitive data through the reduction step of that construct, along the horizontal curly arrow. This section and Section 10.1 later apply two-steps erasure systematically to recover the single-step simulation and then prove noninterference.

7.3 Erasure Function

We proceed to define the erasure function for the pure calculus. Since security labels are given as type annotations, the erasure function is type-driven. We write $\varepsilon_{\ell_A}(t :: \tau)$ for the term obtained by erasing data above the attacker's security level ℓ_A from term t with type τ. In the following, we omit the type annotation when it is either irrelevant or clear from the context. Ground values (e.g., $True$) are unaffected by the erasure function and, for most terms, the function is

$$\varepsilon_{\ell_A}(\langle \Sigma, t :: MAC\ \ell_H\ \tau\rangle) = \begin{cases} \langle \varepsilon_{\ell_A}(\Sigma), \bullet\rangle & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \langle \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(t)\rangle & \text{otherwise} \end{cases}$$

(a) Erasure for configurations.

$$\varepsilon_{\ell_A}(t_s :: Memory\ \ell_H) = \begin{cases} \bullet & \text{if } \ell_H \not\sqsubseteq \ell_A \\ map\ \varepsilon_{\ell_A}\ t_s & \text{otherwise} \end{cases}$$

(b) Erasure for memory.

$$\varepsilon_{\ell_A}(Ref\ n :: Ref\ \ell_H\ \tau) = \begin{cases} Ref\ \bullet & \text{if } \ell_H \not\sqsubseteq \ell_A \\ Ref\ n & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(new\ t :: MAC\ \ell_L\ (Ref\ \ell_H\ \tau)) = \begin{cases} new_\bullet\ \varepsilon_{\ell_A}(t) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ new\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(write\ t_1\ t_2) = \begin{cases} write_\bullet\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2 :: Ref\ \ell_H\ \tau) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ write\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

(c) Erasure for references and memory primitives.

Fig. 16: Erasure for configurations, stores and memory primitives.

applied homomorphically, e.g., $\varepsilon_{\ell_A}(t_1\ t_2 :: \tau) = \varepsilon_{\ell_A}(t_1 :: \tau' \to \tau)\ \varepsilon_{\ell_A}(t_2 :: \tau')$. Figure 15 shows the definition of the erasure functions for the interesting cases. The *content* of a resource of type $Labeled\ \ell_H\ \tau$ is rewritten to $\bullet$ if the label is above the attacker's label, i.e., $\ell_H \not\sqsubseteq \ell_A$, otherwise it is erased homomorphically.[17] Similarly, the erasure function rewrites the argument of *label* to $\bullet$, if it gets labeled above the attacker's level, or erased homomorphically otherwise. Observe that this definition respects the commutativity of the diagram in Figure 14 for rule [Labeл].

Figure 16 shows the erasure function for configurations, stores and memory primitives. Figure 16a presents the erasure for configurations of the form $\langle \Sigma, t\rangle$, which erases the store Σ pointwise, i.e., $\varepsilon_{\ell_A}(\Sigma) = \lambda\ell.\varepsilon_{\ell_A}(\Sigma(\ell))$, and the term t to $\bullet$, if it represents a *sensitive* computation, i.e., if term t has type $MAC\ \ell_H\ \tau$, where ($\ell_H \not\sqsubseteq \ell_A$), and homomorphically otherwise. It is worth pointing out that the erasure of sensitive computations aggressively replaces the whole term with $\bullet$ *only* when considered inside a configuration. When these terms

[17] The special term $\bullet$ can have any type τ. We give the typing rules for the extended calculus in Figure 34 in C.

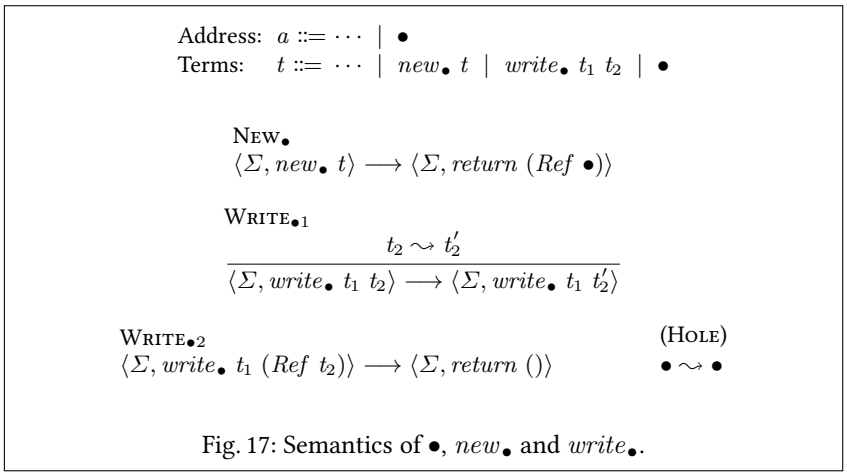


Fig. 17: Semantics of $\bullet$, $new_\bullet$ and $write_\bullet$.

are considered in *isolation*, erasure is applied homomorphically.[18] Intuitively the term alone only *describes* a computation, which executes and performs side-effects only if paired with a store inside a configuration.[19]

References. In Figure 16b, the erasure function rewrites sensitive memories to $\bullet$ and public memories homomorphically. In Figure 16c erasures replaces the address of sensitive references with $\bullet$ and leaves those of public references untouched. The erasure of primitives new and $write$ is non-standard. These primitives perform a *write* effect and only affect memories at least as sensitive as the current computation due to the *no write-down* rule. When these operations change sensitive memories ($\ell_H \not\sqsubseteq \ell_A$), we apply our *two-steps erasure* technique to guarantee lock-step simulation. To do that, the erasure function replaces the constructs new and $write$ with the constructs $new_\bullet$ and $write_\bullet$. These new terms execute according to the rules in Figure 17, which reduce similarly to the original terms, but without performing any write operation. For example, rule [New$_\bullet$] leaves the store Σ unchanged (the argument to $new_\bullet$ is ignored), and simply returns a dummy reference with address $\bullet$. The same principle applies to $write_\bullet$. Rule [Write$_{\bullet 1}$] evaluates the reference and simulates rule [Write$_1$], and rule [Write$_{\bullet 2}$] skips the write operation and simply returns unit to simulate [Write$_2$]. Intuitively, the erased terms leave the memory store unchanged because the original reductions change only parts of the memory above the attacker's security level. We remark that terms $\bullet$, $new_\bullet$ and $write_\bullet$ and their semantics rules are introduced in the calculus due to mere

[18] This is different from the conference version of this work [153], where $\varepsilon_{\ell_A}(MAC\ \ell_H\ \tau :: t) = \bullet$ if $\ell_H \not\sqsubseteq \ell_A$. Erasing these terms homomorphically simplifies the formalization.

[19] Observe that in [153] this was not the case, because rule [Unlabel$_2$] and [Bind$_2$] were given as pure reductions ($\rightsquigarrow$). By separating the pure semantics from the top-level monadic semantics, we simplify also the formalization of applicative functors in Section 10.1.

Terms: $t ::= \cdots \mid \mathit{join}_\bullet\ t$

$$\varepsilon_{\ell_A}(\mathit{join}\ t :: \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} \mathit{join}_\bullet\ \varepsilon_{\ell_A}(t) & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{join}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{Labeled}_\chi\ t :: \mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau) = \begin{cases} \mathit{Labeled}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{Labeled}_\chi\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

(JOIN$_\bullet$)
$\langle \Sigma, \mathit{join}_\bullet\ t\rangle \longrightarrow \langle \Sigma, \mathit{return}\ (\mathit{Labeled}\ \bullet)\rangle$

Fig. 18: Erasure of join and $\mathit{Labeled}_\chi$ and semantics of $\mathit{join}_\bullet$.

technical reasons (as explained above)—they are not part of the surface syntax nor **MAC**.

Join and Exceptions. Using the erasure function, we now study the security guarantees of primitive join and the exception handling primitives, see Figure 18. Primitive join represents a write operation because it creates a separate context where a nested computation executes, thus we apply two-steps erasure. The interesting case is when the nested computation is sensitive ($\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$) and the erasure function replaces join with the special term $\mathit{join}_\bullet$. Then, erasure is performed by rule [JOIN$_\bullet$], which *immediately* returns a dummy labeled value ($\mathit{Labeled}\ \bullet$) and the store unchanged. Intuitively, a sensitive computation cannot change public memories (*no write-down*), and join protects the final result inside a labeled value, which the attacker cannot unlabel (*no read-up*). As a result, this rule captures the observational power of an attacker that runs a *terminating* sensitive computation. What about nested sensitive computations that fail with an exception? Since primitive join hides these exceptions inside a sensitive labeled value as well (rule [JOIN$_\chi$] in Figure 11), also this condition remains unobservable to the attacker. Formally, the erasure function rewrites the exception t inside a sensitive labeled value $\mathit{Labeled}_\chi\ t$ to $\bullet$ and then *masks* its exceptional nature, by replacing the constructor $\mathit{Labeled}_\chi$ with $\mathit{Labeled}$, thus ensuring that rule [JOIN$_\bullet$] simulates rule [JOIN$_\chi$] as well. Crucially, we have the freedom of choosing this definition without breaking *simulation*, because no other construct can detect, either explicitly or implicitly, the difference. For instance, consider rule [UNLABEL$_\chi$] from Figure 11, which rethrows exceptions hidden inside a labeled value. If the labeled value is above the attacker, then the computation that executes $\mathit{unlabel}$ must also be at least as sensitive as the labeled value due to the typing rules (*no read-up*). Then, primitive $\mathit{unlabel}$ gets rewritten to $\bullet$ and the step is simulated by rule [HOLE] instead. As a result of that, and unlike the approach taken by Stefan et al. in [143], there are no sensitive labeled exceptions in erased terms.

7.4 Discussion

Term Erasure. In this work, we prove single-step simulation directly over the small-step reduction relation using our novel two-steps erasure technique. In contrast, previous works have proved simulation by relating the small-step relation (upper part in Figure 14) with an ℓ_A-indexed small-step relation of the form $c \longrightarrow_{\ell_A} \varepsilon_{\ell_A}(c')$, which applies erasure at every reduction step [55, 76, 124, 141, 143, 145]. These works reason about dynamic IFC languages featuring dynamic security policies (e.g., labels are run-time terms), which complicate reasoning about sensitive data statically, and thus require a different simulation relation. In contrast, **MAC** does not need such an auxiliary construction because it enforces security statically, i.e., labels are not terms but type-level annotations, and therefore known before execution. In this light, our erasure function can safely erase sensitive data from labeled values based on their type. Our small-step semantics satisfies type-preservation, thus execution does not change the type-level label annotations and we can prove single-step simulation without the need of a special indexed small-step relation.

Masking Sensitive Exceptions. Previous work also studied the security guarantees of exception handling primitives using the erasure function [143]. However, the approach proposed there preserves the exceptional state of labeled exceptions and only removes their sensitive content. In contrast, our approach fully masks sensitive exceptions in erased programs. Intuitively, sensitive exceptions never arise in erased programs because our erasure function always replaces them with unexceptional labeled values. The single-step simulation property guarantees the soundness of our rewriting approach. In particular, primitive $unlabel$—the only construct that can distinguish between exceptional and unexceptional labeled values—gets also erased as explained above.

Memory. It is known that dealing with dynamic memory allocation complicates noninterference proofs [15,53]. Banerjee et al. employ a non-split memory model, where computations at different security levels share the same memory address space [15]. In that model, sensitive allocations do affect the global memory layout, which in turn affects the address of public references. Proving noninterference with this memory model requires establishing a bijection between public memory addresses of two executions and considering equality of public references up to such notion [15]. Unsurprisingly, proving noninterference with a non split-memory model requires special attention: for instance, we have identified problems with the proofs in manuscripts and articles related to **LIO** [143, 145]. (We refer interested readers to Appendix B of our conference version [153] for details.) To avoid this inconvenience, our model compartmentalizes the memory into isolated labeled segments, so that allocations in one segment does not affect the others, similar to other works [55, 145]. The fact that GHC's memory is non-split does not compromise our security guarantees, because references are part of **MAC**'s internals and they cannot be inspected or deallocated explicitly. However, this memory model assumes infinite memory,

e.g., reference allocation never fails. This assumption is unrealistic: physical resources such as memory are finite.[20] As a result, **MAC** is vulnerable to the memory-exhaustion covert channel, which leaks secret information with the same bandwidth of the termination covert channel [4].

In the conference version of this work [153], we have explored an alternative way to prove *single-step* simulation for terms new and $write$ consists in extending the semantics of memory operations to node $\bullet$, i.e., by defining $|\bullet| = \bullet$ and $\bullet[\bullet \mapsto t] = \bullet$. Thanks to two-steps erasure, we can prove simulation as we did here, without recurring to a non-standard memory semantics.

7.5 Progress-Insensitive Non-interference

The sequential calculus that we have presented satisfies *progress-insensitive noninterference*. The proof of this result is based on two fundamental properties: *single-step simulation* and *determinancy* of the small step semantics. In the following, we assume well-typed terms.

Proposition 1 (Single-step Simulation)
Given two configurations c_1 and c_2, if $c_1 \longrightarrow c_2$, then $\varepsilon_{\ell_A}(c_1) \longrightarrow \varepsilon_{\ell_A}(c_2)$.

Proof (Sketch). By induction on the reduction steps and typing judgment. Sensitive computations are simulated by transition $\langle \Sigma, \bullet \rangle \longrightarrow \langle \Sigma, \bullet \rangle$, obtained by lifting rule [Hole] with [Pure]. Non-sensitive computations are simulated by the same rule that performs the non-erased transition, except when it involves some *sensitive write* operations, e.g., in rules [New, Write$_1$, Write$_2$, Join, Join$_\chi$], which are simulated by rules [New$_\bullet$, Write$_\bullet$1, Write$_\bullet$2, Join$_\bullet$].

Proposition 2 (Determinancy) *If $c_1 \longrightarrow c_2$ and $c_1 \longrightarrow c_3$ then $c_2 \equiv c_3$.*

Proof. By standard structural induction on the reductions.[21]

Before proving progress-insensitive noninterference, we define ℓ_A-equivalence for configurations.

Definition 1 (ℓ_A-equivalence). *Two configurations c_1 and c_2 are indistinguishable from an attacker at security level ℓ_A, written $c_1 \approx_{\ell_A} c_2$, if and only if $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$.*

Using Propositions 1 and 2, we show that our semantics preserves ℓ_A-equivalence.

[20] Unrestricted access to shared resources often constitutes a covert channel in concurrent systems. The resources can be either hardware (e.g., physical memory, caches [140] and multi-core processors) or software, such as components of the run-time system of a high-level language. These include the scheduler [125, 126] and the garbage collector [107] or the language evaluation strategy such as lazy evaluation [31, 151].

[21] Symbol $\equiv$ denotes equivalence up to alpha equivalence in the calculus with named variables. In our mechanized proofs we use Bruijn indexes and we obtain syntactic equality.

Proposition 3 ($\approx_{\ell_A}$ Preservation)
If $c_1 \approx_{\ell_A} c_2$, $c_1 \longrightarrow c_1'$, *and* $c_2 \longrightarrow c_2'$, *then* $c_1' \approx_{\ell_A} c_2'$.

By repeatedly applying Proposition 3, we prove progress-insensitive noninterference.

Theorem 1 (PINI) *If* $c_1 \approx_{\ell_A} c_2$, $c_1 \Downarrow c_1'$ *and* $c_2 \Downarrow c_2'$, *then* $c_1' \approx_{\ell_A} c_2'$.

8 Concurrency

Real world software relies on concurrency for scaling and reacting to external inputs. Examples of concurrent applications abound, including graphical user interfaces, web servers, chat rooms and distributed databases. Unfortunately, extending **MAC** with concurrency is not trivial. In the following, we show that running several computations simultaneously give attackers new means to break the security guarantees of **MAC**. The rest of this section extends **MAC** with *concurrency* and adds secure *thread synchronization* primitives, which allow developers to write secure-by-construction concurrent programs.

8.1 Termination Attack

The previous section shows that **MAC** guarantees security only for terminating programs. The key observation is that primitive $join$ leaks information via non-termination. Intuitively, a secret computation embedded in a public computation via $join$ may loop and thus suppress public side-effects of the outer computation.[22] In particular, if the embedded computation loops depending on secret information, then the control flow of the program never returns back to the public computation and the absence of subsequent public side-effects leaks information. To illustrate this point, consider the program $leak\ n\ secret$ in Figure 19, which leaks the value of the nth bit of the secret through a termination attack. Intuitively, the public program starts a secret computation, which extracts the secret ($bits \leftarrow unlabel\ secret$), and then loops if the nth bit of the secret is 1 (**when** $(bits\ !!\ n)\ loop$), or otherwise returns to the public computation, which then prints the integer n on the screen.[23] As a result, the program $leak\ 0\ secret$ prints 0 *only if* the first bit of $secret$ is 0; otherwise it loops and it does not produce any public effect. Similarly, program $leak\ 1\ secret$ leaks the second bit of $secret$, program $leak\ 2\ secret$ leaks the third bit of it and so on.

Even though guaranteeing security only for terminating programs might be acceptable for sequential programs, the next example demonstrates that this security condition is too weak for concurrent systems.

[22] If the physical execution time of a program depends on the value of the secret, then an attacker with an arbitrary precise stopwatch can deduce information about the secret by timing the program. This covert channel is known as the *external timing covert channel* [24, 41]. This article does not address the external timing covert channel, which is a harder problem and for which mitigation techniques exist [5, 168, 169].

[23] **MAC** considers printing on the screen a *public* side-effect, i.e., $print^{\text{MAC}} :: Int \rightarrow MAC\ L\ ()$.

```
leak :: Int → Labeled H Secret → MAC L ()
leak n secret = do
  join^MAC (do bits ← unlabel secret
               when (bits !! n) loop
               return ())
  print^MAC n
```

Fig. 19: Termination leak.

```
magnify :: Labeled H Secret → MAC L ()
magnify secret = for [0 .. |secret|] (λn → fork (leak n secret))
```

Function *magnify* leaks the whole secret by spawning as many threads as bits in the secret, i.e., $|secret|$, where each thread runs the one-bit attack described above and n matches the bit being leaked (e.g., $n = 0$ for the first bit, $n = 1$ for the second one, etc.). This example shows that concurrency magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets [141], which permits to leak any secret efficiently.[24]

To securely support concurrency, **MAC** forces programmers to decouple MAC computations with sensitive labels from those performing observable side-effects—an approach also taken in LIO [141]. As a result, non-terminating computations based on secrets cannot affect the outcome of public events. To enforce this separation, **MAC** replaces *join* by *fork*:

$$fork :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow MAC\ \ell_{\mathrm{H}}\ () \rightarrow MAC\ \ell_{\mathrm{L}}\ ()$$

Informally, it is secure to spawn sensitive computations (of type $MAC\ \ell_{\mathrm{H}}\ ()$) from less sensitive ones (of type $MAC\ \ell_{\mathrm{L}}\ ()$) because that decision depends on data at level ℓ_{L}, which is no more sensitive ($\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$). From now on, we call *sensitive* (*non-sensitive*) *threads* those executing MAC computations with a label non-observable (observable) to the attacker. In the two-point lattice, for example, threads running $MAC\ H\ ()$ computations are sensitive, while those running $MAC\ L\ ()$ are observable by the attacker. In order to prove that the API above is secure, the next section formalizes the semantics of concurrent **MAC**.

8.2 Semantics

This section extends the sequential calculus from Section 3 with concurrency, see Figure 20. Figure 20a introduces *global configurations* of the form $\langle \omega, \Sigma, \Phi \rangle$

[24] Furthermore, the presence of threads introduce the *internal timing covert channel* [138], a channel that gets exploited when, depending on secrets, the timing behavior of threads affect the order of events performed on public-shared resources. Since the same countermeasure closes both the internal timing and termination covert channels, we focus on the latter.

Scheduler State:	ω
Pool Map :	$\Phi \in (\ell : Label) \to (Pool\ \ell)$
Thread Pool ℓ:	$t_s ::= [\,] \mid t : t_s$
Configuration:	$c ::= \langle \omega, \Sigma, \Phi \rangle$
Sequential Event ℓ:	$s ::= \varnothing \mid fork(t :: \tau)$
Concurrent Event ℓ:	$e ::= Step \mid Stuck \mid Done \mid Fork\ \ell\ n$
Terms:	$t ::= \cdots \mid fork\ t$

(a) Syntax of concurrent calculus.

$$\frac{\Phi(\ell)[n] = t_1 \qquad \langle \Sigma_1, t_1 \rangle \longrightarrow_s \langle \Sigma_2, t_2 \rangle \qquad \omega_1 \xrightarrow{(\ell,n,e)} \omega_2}{\langle \omega_1, \Sigma_1, \Phi \rangle \hookrightarrow \langle \omega_2, \Sigma_2, \Phi(\ell)[n \mapsto t_2] \rangle}$$

(b) Scheme rule for concurrent semantics.

(SFork)
$$\langle \Sigma, fork\ t \rangle \longrightarrow_{fork(t)} \langle \Sigma, return\ () \rangle$$

(Bind$_1$)
$$\frac{\langle \Sigma, t_1 \rangle \longrightarrow_s \langle \Sigma', t_1' \rangle}{\langle \Sigma, t_1 \gg\!= t_2 \rangle \longrightarrow_s \langle \Sigma', t_1' \gg\!= t_2 \rangle}$$

(Catch$_1$)
$$\frac{\langle \Sigma, t_1 \rangle \longrightarrow_s \langle \Sigma', t_1' \rangle}{\langle \Sigma, catch\ t_1\ t_2 \rangle \longrightarrow_s \langle \Sigma', catch\ t_1'\ t_2 \rangle}$$

(c) Decorated Sequential Semantics (interesting rules).

Fig. 20: Calculus with concurrency.

composed by an *abstract* scheduler state ω, a store Σ and a pool map Φ. Threads are secure computations of type $MAC\ \ell\ ()$ and are organized in isolated thread pools according to their security label. A pool t_s in the category $Pool\ \ell$ contains threads at security level ℓ and is accessed exclusively through the pool map. For manipulating thread pools and pool maps, we use the same notation defined for stores and memories. Term $fork\ t$ spawns thread t and replaces $join$ in the calculus. Without $join$, constructor $Labeled_\chi$ becomes redundant and is also removed. The concurrent calculus includes also synchronization primitives [123], which are formalized in Appendix B.

Dynamics. Relation $c_1 \hookrightarrow c_2$ denotes a concurrent reduction step, where configuration c_1 reduces to configuration c_2. Figure 20b shows the *scheme rule* for $c_1 \hookrightarrow c_2$ (all concurrent reductions follow this pattern), the concrete rules

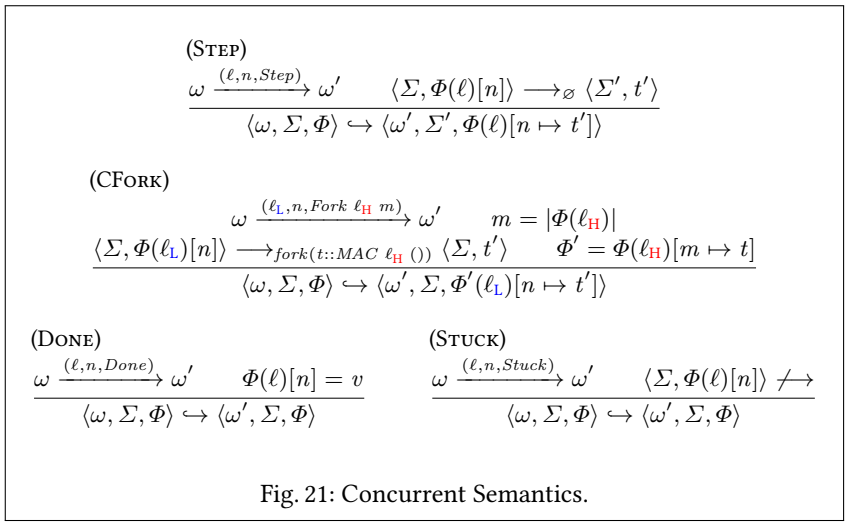


Fig. 21: Concurrent Semantics.

are presented later in Figure 21. In the scheme rule, the relation $\omega_1 \xrightarrow{(\ell,n,e)} \omega_2$ represents a transition in the scheduler, which decides to run thread identified by (ℓ, n), based on the initial state ω_1. Then, the scheme rule retrieves that thread from the configuration ($\Phi(\ell)[n] = t_1$) and runs it through the *decorated* sequential semantics (explained below). The scheme rules uses *concurrent events* to inform the abstract scheduler about the evolution of the global configuration, so that it can realize concrete scheduling policies and update its state accordingly. Specifically, event $Step$ signals that the running thread has stepped, event $Fork\ \ell\ n$ informs the scheduler that the running thread has forked a new thread identified by (ℓ, n), and event $Done$ and $Stuck$ signal that the running thread has terminated and got stuck (e.g., on a synchronization variable), respectively. In the concrete rules, the *sequential event* s from the decorated semantics, i.e., $\langle \Sigma, t_1\rangle \longrightarrow_s \langle \Sigma, t_2\rangle$, determines what concurrent event ($Step$ or $Fork$) gets passed to the scheduler. Lastly, the scheme rule updates the configuration with the resulting scheduler state, memory and thread pool, i.e., $\langle \omega_2, \Sigma_2, \Phi(\ell)[n \mapsto t_2]\rangle$.

Decorated Semantics. Figure 20c shows the interesting rules of the decorated semantics. Rule [SFork] is the only rule that explicitly generates event $fork(t)$, rules [Bind$_1$,Catch$_1$] simply propagate the event generated in the premise, and all the other rules generate the empty event $\varnothing$.

Concurrent Semantics. Figure 21 shows the concrete concurrent semantics, wherein each rule generates the appropriate concurrent event for the scheduler based on the state of the running thread and the sequential event. Rule [Step] sends event $Step$ to the scheduler, because the thread generates sequential event $\varnothing$. In rule [CFork], the running thread forks a child thread (event $fork(t :: MAC\ \ell_H\ ())$, which is placed in the thread pool $\Phi(\ell_H)$ in the *free* position $m = |\Phi(\ell_H)|$. Then, the rule sends event $Fork\ \ell_H\ m$ to the scheduler and

Scheduler State: $\omega ::= (\ell, n) : \omega \mid [\,]$

$$(\ell, n) : \omega \xrightarrow{(\ell, n, \mathit{Step})}_{RR} \omega +\!\!+ [(\ell, n)] \qquad (\ell, n) : \omega \xrightarrow{(\ell, n, \mathit{Stuck})}_{RR} \omega +\!\!+ [(\ell, n)]$$

$$(\ell_{\mathrm{L}}, n_1) : \omega \xrightarrow{(\ell_{\mathrm{L}}, n_1, \mathit{Fork}\ \ell_{\mathrm{H}}\ n_2)}_{RR} \omega +\!\!+ [(\ell_{\mathrm{H}}, n_2), (\ell_{\mathrm{L}}, n_1)]$$

$$(\ell, n) : \omega \xrightarrow{(\ell, n, \mathit{Done})}_{RR} \omega$$

Fig. 22: Round-robin scheduler.

updates the thread pool with the child and the parent thread. Rule [Done] sends event Done to the scheduler because the scheduled thread has terminated, i.e., term $\Phi(\ell)[n] = v$ is a value, and leaves the store and the pool map unchanged (the semantics does not remove terminated threads from the thread pool). In rule [Stuck], the notation $\Phi(\ell)[n]\rangle \not\longrightarrow$ means that the thread is stuck, i.e., it is not a value and cannot reduce, thus the rule sends event Stuck to the scheduler.

The concurrent semantics presented above features an abstract scheduler, which allows to study the security guarantees of **MAC** in isolation from the concrete scheduling policy. Since the implementation of **MAC** relies on GHC's round-robin scheduler for scheduling threads [86], the next section instantiates the abstract scheduler with a concrete round-robin scheduler.

8.3 Round Robin Scheduler

Figure 22 presents a model of a round-robin scheduler with time-slot of one step and state consisting of a queue of thread identifiers to be scheduled. The queue ω consists concretely of a list of thread identifiers, where the first element in the list identifies the next thread in the schedule. The reduction rules formalize the round-robin scheduling policy. The rules have form $\omega_1 \xrightarrow{(\ell, n, e)} \omega_2$, where (ℓ, n) identifies the next running thread and concurrent event e informs the scheduler about its execution, similar to the abstract relation used in the concurrent semantics above. The rules reduce only with a non-empty queue so that the scheduler steps only if there is a thread to schedule. Then, the scheduler state determines the running thread, i.e., the next thread in the queue, and the rules update the state uniquely based on the event. This design choice allows defining the scheduling policy in isolation from the concurrent semantics. Event Step and Stuck inform the round robin scheduler that the running thread has used up its time slot, thus the scheduler moves it to the back of the queue. Similarly, event $\mathit{Fork}\ \ell_{\mathrm{H}}\ n_2$ informs the scheduler that the running thread (ℓ_{L}, n_1) has spawned child thread (ℓ_{H}, n_2), which is added to the back of the queue with its parent. If the running thread terminates, event Done, the scheduler simply removes it from the queue, which then contains only alive threads.

Before proving security for the concurrent calculus, we add a functor algebraic structure to labeled data, in a way that it can be processed through classic functional programming patterns that do not incur in security checks.

9 Flexible Labeled Values

This section extends the API of labeled values with new operations that allow to perform *pure* (side-effect free) computations with labeled data. These primitives make labeled data *flexible* because they allow programs to process sensitive data without using $unlabel$ and $label$, and thus avoid the corresponding *no read-up* and *no write-down* restrictions. Furthermore, these primitives foster a functional programming style—they also avoid the use of other imperative operations, such as $join$ or $fork$ in sequential and concurrent **MAC**, respectively. Section 9.1 gives a broad description of the API of flexible labeled data and Section 9.2 argues for its flexibility with several examples. Lastly, Section 9.3 adds these primitives to our calculus and discusses some subtleties when adding these features to languages with different evaluation strategies, wherein a naive implementation could open the termination channel.

9.1 Functors and Relabeling

Intuitively, a functor is a container-like data structure which provides a method called $fmap$ that applies (maps) a function over its contents, while preserving its structure. Lists are the most canonical example of a functor data structure. For lists, method $fmap$ corresponds to the standard function map, which applies a function to each element of a list, e.g., $fmap\ (+1)\ [1,2,3] \equiv [2,3,4]$. Adding a functor structure to labeled data simply requires implementing a similar method $fmap$, so that, for example, executing $fmap\ (+1)\ (Labeled\ 42)$ produces $Labeled\ 43$. Together with $fmap$, **MAC**'s API for flexible labeled data includes the following methods:

$$
\begin{aligned}
&fmap :: (a \to b) \to Labeled\ \ell\ a \to Labeled\ \ell\ b\\
&\langle * \rangle :: Labeled\ \ell\ (a \to b) \to Labeled\ \ell\ a \to Labeled\ \ell\ b\\
&relabel :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow Labeled\ \ell_{\mathrm{L}}\ a \to Labeled\ \ell_{\mathrm{H}}\ a
\end{aligned}
$$

Notice the type-signature of $fmap$, which allows processing labeled data at any security level ℓ with an arbitrary *pure* function. Intuitively, this primitive is secure because Haskell's type system ensures that the function is side-effect free and the result remains labeled at the same security level ℓ. To aggregate data at possibly different security levels **MAC** provides primitives $\langle * \rangle$ and $relabel$. Infix operator $\langle * \rangle$ supports function application within labeled values. It allows applying a function wrapped in a labeled value ($Labeled\ \ell\ (a \to b)$) to a labeled argument ($Labeled\ \ell\ a$) to produce a labeled value that contains the result of the function application ($Labeled\ \ell\ b$). Primitive $relabel$ upgrades the security level of labeled data—a feature useful to "lift" data at different security levels to a common upper bound label and then combine it through operator $\langle * \rangle$.

Discussion. In functional programming, operator $\langle * \rangle$ is part of the *applicative functor* [89] interface, which in combination with $fmap$, is used to map functions over functors. Note that if labeled values were full-fledged applicative

```
isShort :: Labeled H String → MAC L (Labeled H Bool)
isShort lpwd = do
  join (do
    pwd ← unlabel lpwd
    return (|pwd| ⩽ 5))
```

Fig. 23: The program *isShort* "simply" checks if the password is short.

functors, our API would also include the primitive $pure :: a \to Labeled\ \ell\ a$. This primitive brings arbitrary values into labeled values, which might break the security principles enforced by **MAC**. Instead of *pure*, **MAC** centralizes the creation of labeled values in the primitive *label*. Observe that, by using *pure*, a programmer could write a program of type $MAC\ H\ (Labeled\ L\ a)$, where the resulting labeled data is sensitive rather than public. We argue that this situation ignores the no-write down principle, which might bring confusion among users of the library. More importantly, freely creating labeled values is not compatible with the security notion of *cleareance*, where secure computations have an upper bound on the kind of sensitive data they can observe and generate. This notion becomes useful to address certain covert channels [166] as well as poison-pill attacks [59]. While **MAC** does not yet currently support cleareance, it is an interesting direction for future work.

9.2 Examples

The API of flexible labeled values (i.e., functions *fmap*, $\langle * \rangle$, and *relabel*) allows functional programmers to manipulate sensitive data concisely. Without primitives such as *fmap*, writing even simple functions over labeled data becomes cumbersome. For example, consider the program *isShort* in Figure 23, which simply checks if a secret password is less than 5 characters long. Without *fmap*, the program has to use *join* just to lift a secret computation that unlabels the password and computes its length. Because of the security restrictions of **MAC**, the program is longer than it should and carries an imprecise type. In fact, $isShort\ lpwd$ has a monadic type, i.e., $MAC\ L\ (Labeled\ H\ Bool)$, even though the program does not perform any IO, but just because it relies on *join* to convert a secret computation into labeled data. Even worse, concurrent **MAC** does not feature the primitive *join* and the whole program must be completely restructured.[25] Compare the program from Figure 23 with the following program that uses *fmap* instead:

```
isShort' :: Labeled H String → Labeled H Bool
isShort' lpwd = fmap (λpwd → |pwd| ⩽ 5) lpwd
```

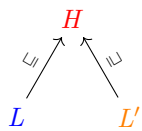
[25] Forking a thread does not help because threads cannot return their result directly for security reasons—doing so would reintroduce the termination channel!

The function $isShort'$ computes the same result of $isShort$, but it does so in only one line of code and it carries an accurate type (it does not involve any MAC computation). Furthermore, this program works for both sequential and concurrent **MAC**. Notice that $isShort'$ does not rely on any security primitive other than $fmap$. In particular, the lambda expression does not need to unlabel the password—it has free access to the *unlabeled* password (pwd)—nor it needs to label the result explicitly—the type signature of $fmap$ ensures that the result remains at the same security level.

The strength of a password is often measured by combining various metrics, such as the length and the presence of special characters and digits. Suppose now that some third-party API function provides such syntactic checks in the form of the following **MAC** labeled function $isWeak$:

$$isWeak :: Labeled\ L'\ (String \to Bool)$$

Haskell type system guarantees that the function inside the labeled value cannot leak data: it has pure type $String \to Bool$. However the third party has labeled the function with its own label L' in order to restrict access to it. To keep the code of our password-checker isolated from the third-party code, while still being able to make use of it, we incorporate the new label L' into the system and modify the security lattice as follows:



The lattice reflects our mistrust over the third-party code by making L and L' incomparable elements. Thanks to **MAC**'s security guarantees, we can safely run the third-party mistrusted code with our user's secret password:

$$\begin{aligned} &secure :: Labeled\ H\ String \to Labeled\ H\ Bool \\ &secure\ lpwd = (relabel\ isWeak)\ \langle * \rangle\ lpwd \end{aligned}$$

In the code above, primitive $relabel$ upgrades the function $isWeak$ to security level H (observe that $L' \sqsubseteq H$ in the lattice). Then, the applicative functor operator $\langle * \rangle$ applies the relabeled function to the secret password ($lpwd$) and gives the result, protected at security level H.

9.3 Semantics

In order to study the security guarantees of the API of flexible labeled data, we incorporate these primitives in our calculus. Figure 24 adds nodes $fmap\ t_1\ t_2$, $t_1\ \langle * \rangle\ t_2$ and $relabel\ t$ to the category of terms. Rule [Fmap] relies on the fact

Terms: $t ::= \cdots \mid \mathit{fmap}\ t_1\ t_2 \mid t_1\ \langle * \rangle\ t_2 \mid \mathit{relabel}\ t$

(FMAP)
$\mathit{fmap}\ t_1\ t_2 \rightsquigarrow (\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2$

$(\langle * \rangle_1)$
$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ \langle * \rangle\ t_2 \rightsquigarrow t_1'\ \langle * \rangle\ t_2}$$

$(\langle * \rangle_2)$
$$\frac{t_2 \rightsquigarrow t_2'}{(\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2 \rightsquigarrow (\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2'}$$

$(\langle * \rangle_3)$
$(\mathit{Labeled}\ t_1)\ \langle * \rangle\ (\mathit{Labeled}\ t_2) \rightsquigarrow \mathit{Labeled}\ (t_1\ t_2)$

(RELABEL$_1$)
$$\frac{t \rightsquigarrow t'}{\mathit{relabel}\ t \rightsquigarrow \mathit{relabel}\ t'}$$

(RELABEL$_2$)
$\mathit{relabel}\ (\mathit{Labeled}\ t) \rightsquigarrow \mathit{Labeled}\ t$

Fig. 24: Calculus with flexible labeled values.

that applicative functors generalize simple functors and reduces $\mathit{fmap}\ t_1\ t_2$ to $(\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2$, by simply lifting the function to labeled value. Then, rule $[\langle * \rangle_1]$ and $[\langle * \rangle_2]$ evaluate the function and the argument to a labeled value, respectively. Lastly, rule $[\langle * \rangle_3]$ applies the function to the argument and wraps the result in a labeled value. Rule [RELABEL$_1$] evaluates the argument to a labeled value and then rule [RELABEL$_2$] upgrades its label. Since labels are only static type-annotations, *relabel* leaves the content of the labeled value unchanged. We remark that these primitives are secure both in the concurrent and sequential calculus. Appendix A presents additional evaluation rules that deal with exceptional labeled values, i.e., constructor $\mathit{Labeled}_\chi$, which is present only in the sequential calculus.

Discussion. Extending **MAC** with the API of flexible labeled values presented above might seem insecure at first sight. After all, these primitives (e.g., *fmap* and $\langle * \rangle$) allow *arbitrary* pure functions to freely access secret data from public contexts as well. Crucially, even functions deemed "pure" by Haskell type-system can exhibit certain effects, such as non-termination. Then, it is reasonable to study whether these primitives are vulnerable to termination attacks. Consider the following variant of the termination attack from Figure 19, which attempts to leak the n-th bit of the secret through *fmap*.

```
leak :: Int → Labeled H Secret → MAC L ()
leak n secret =
  let result = fmap loopOn secret in
    print^MAC n
  where loopOn = λbits → if (bits !! n) then loop else bits
```

Function $leak$ applies function $loopOn$ on the secret using $fmap$ and then performs a public side-effect, i.e., $print^{\text{MAC}}\ n$, which prints the number n on the screen. As seen previously, the code leaks information if it can suppress the public side-effect based on sensitive information, i.e., the value of the nth bit of the secret. Interestingly, the evaluation strategy of the language determines the outcome of the attack. With *call-by-value*, the attack succeeds: primitive $fmap$ applies function $loopOn$ to the secret *eagerly*, and thus might trigger the loop and suppress the public print. However, with *call-by-name*, the attack fails: the function application, i.e., $fmap\ loopOn\ secret$, is delayed—the result is not needed in the rest of the program—and the program always prints number n on the screen. In call-by-name languages, forcing the loop requires *unlabeling* the result: something that only a secret computation at security level H can do! Nevertheless, also call-by-value languages can support the flexible labeled value API securely, but the implementation requires more care to avoid introducing the termination channel from above. To avoid leaking, the key insight is to make the $Labeled$ data-type *non-strict*, by means of explicit suspension and forcing functions. For example, the following definition of $Labeled$ is secure even in strict languages with $fmap$.

$$\textbf{data}\ Labeled\ \ell\ a = Labeled\ (() \to a)$$

With this definition, a labeled value does not store a piece of data directly, but hides it behind a suspension function of type $() \to a$, which must be *forced*, i.e., applied to unit, in order to extract the data. Thanks to the the suspension, labeled values and $fmap$ behave non-strictly and do not leak information through the termination channel.

After extending our calculus with concurrency and flexible labeled data, we now proceed to study the security guarantees of our concurrent model.

10 Soundness of Concurrent Calculus

This section presents a scheduler-parametric progress-sensitive noninterference proof for the concurrent calculus. Firstly, Section 10.1 extends the erasure function for the new features (i.e., concurrency and flexible labeled values). Then, to obtain a scheduler-parametric noninterference proof, Section 10.2 formalizes the properties that concrete schedulers must satisfy to preserve security of the whole model. In particular, the noninterference proof is valid for deterministic schedulers which fulfill progress and noninterference themselves, i.e., schedulers cannot leverage sensitive information in threads to determine what to schedule next. Section 10.3 proves the scheduler-parametric progress-sensitive noninterference theorem and derives security for **MAC** by simply instantiating the main theorem with the round-robin scheduler.

10.1 Erasure Function

Figure 25 shows the erasure function for the concurrent calculus. The erasure function for concurrent configurations of the form $\langle \omega, \Sigma, \Phi \rangle$ is per component

$$\varepsilon_{\ell_A}(\langle \omega, \Sigma, \Phi \rangle) = \langle \varepsilon_{\ell_A}(\omega), \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Phi) \rangle$$

(a) Erasure for concurrent configuration.

$$\varepsilon_{\ell_A}(t_s :: Pool\ \ell_H) = \begin{cases} \bullet & \text{if } \ell_H \not\sqsubseteq \ell_A \\ map\ \varepsilon_{\ell_A}\ t_s & \text{otherwise} \end{cases}$$

(b) Erasure for thread pool.

$$\varepsilon_{\ell_A}(fork\ t) = \begin{cases} fork_\bullet\ \varepsilon_{\ell_A}(t :: MAC\ \ell_H\ ()) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ fork\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

(c) Erasure of fork.

$$\varepsilon_{\ell_A}(fork(t :: MAC\ \ell_H\ ())) = \begin{cases} fork_\bullet(\varepsilon_{\ell_A}(t)) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ fork(\varepsilon_{\ell_A}(t)) & \text{otherwise} \end{cases}$$

(d) Erasure for *sequential* fork event.

$$\varepsilon_{\ell_A}(Fork\ \ell_H\ n) = \begin{cases} Step & \text{if } \ell_H \not\sqsubseteq \ell_A \\ Fork\ \ell_H\ n & \text{otherwise} \end{cases}$$

(e) Erasure for *concurrent* fork event.

Fig. 25: Erasure function for concurrent calculus.

(Figure 25a). The erasure of the abstract scheduler state ω is scheduler-specific and the erasure of the pool map Φ is pointwise, i.e., $\varepsilon_{\ell_A}(\Phi) = \lambda\ell.\varepsilon_{\ell_A}(\Phi(\ell))$, just like for the store Σ. Then, the erasure function rewrites secret thread pools to node $\bullet$ and is applied homomorphically for public thread pools, just like for memories (Figure 25b). Since primitive $fork$ performs a *write* effect (it adds a new thread to a thread pool), Figure 25c applies the *two-steps erasure* technique. In particular, the erasure function replaces $fork$ with a new primitive $fork_\bullet$, when it forks a secret thread ($\ell_H \not\sqsubseteq \ell_A$), or is applied homomorphically to the child thread otherwise. Similarly, the erasure function rewrites the sequential event $fork(t)$ to $fork_\bullet(\varepsilon_{\ell_A}(t))$, if the child thread is secret (25d), or follows homomorphically as well otherwise. The empty event $\varnothing$ is instead left unchanged by the erasure function, i.e., $\varepsilon_{\ell_A}(\varnothing) = \varnothing$. Erasure for concurrent events is in-

Sequential Events: $s ::= \cdots \mid \mathit{fork}_\bullet(t)$
Terms: $t ::= \cdots \mid \mathit{fork}_\bullet\ t$

(SFork$_\bullet$)
$\langle \Sigma, \mathit{fork}_\bullet\ t\rangle \longrightarrow_{\mathit{fork}_\bullet(t)} \langle \Sigma, \mathit{return}\ ()\rangle$

(CFork$_\bullet$)
$$\frac{\omega \xrightarrow{(\ell_L, n, \mathit{Step})} \omega' \qquad \langle \Sigma, \Phi(\ell_L)[n]\rangle \longrightarrow_{\mathit{fork}_\bullet(t)} \langle \Sigma, t'\rangle}{\langle \omega, \Sigma, \Phi\rangle \hookrightarrow \langle \omega', \Sigma, \Phi(\ell_L)[n \mapsto t']\rangle}$$

Fig. 26: Sequential and concurrent semantics of $\mathit{fork}_\bullet$.

teresting. In particular, in order to hide the number of secret threads from the scheduler, the erasure function masks secret fork events, e.g., event $\mathit{Fork}\ \ell_H\ n$ where $\ell_H \not\sqsubseteq \ell_A$, and replaces them with event Step (Figure 25e). All the other concurrent events are not affected by the erasure function.

Dynamics. In the sequential calculus, decorated rule [SFork$_\bullet$] evaluates primitive $\mathit{fork}_\bullet$ and generates event $\mathit{fork}_\bullet(t)$ (Figure 26), so that it simulates rule [SFork] from Figure 20c when forking a secret thread. Similarly, the new concurrent rule [CFork$_\bullet$] detects when a public thread forks a secret thread (event $\mathit{fork}_\bullet(t)$) and simulates rule [CFork] from Figure 21. The rule ignores the child thread, which is not added to the thread pool, and sends event Step to the scheduler instead.

Context-Aware Erasure. A common challenge when reasoning about security of IFC libraries is that the sensitivity of a term may depend on context where it gets used. For example, consider the primitive $\mathit{relabel}$, which upgrades the security level of a labeled term. A public number, e.g., $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int}$, should be treated as secret when it appears in the context of $\mathit{relabel}$, e.g., $\mathit{relabel}\ (\mathit{Labeled}\ 42) :: \mathit{Labeled}\ H\ \mathit{Int}$. Doing otherwise, i.e., erasing the term homomorphically, breaks the single-step simulation property because sensitive data produced by $\mathit{relabel}$ remains *after* erasure. For example, with homomorphic erasure, $\varepsilon_L(\mathit{relabel}\ (\mathit{Labeled}\ 42) :: \mathit{Labeled}\ H\ \mathit{Int}) = \mathit{relabel}\ \varepsilon_L(\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int})$, which reduces along the orange path in the simulation diagram from Figure 14 to $\mathit{Labeled}\ 42 \not\equiv \mathit{Labeled}\ \bullet$, obtained along the cyan path by $\varepsilon_L(\mathit{Labeled}\ 42 :: \mathit{Labeled}\ H\ \mathit{Int})$, thus breaking commutativity of rule [Relabel$_2$]. Then, one might be tempted to stretch the definition of the erasure function to accommodate the problematic cases above. Unfortunately, this approach does not scale: the new erasure function will necessarily break the simulation diagram of some other rule. We support this statement by showing that this is the case for an arbitrary erasure function ε'_L that satisfies the simulation diagram for rule [Relabe$_2$] Observe that the new erasure function should behave differently for public labeled values embedded in $\mathit{relabel}$, including $\mathit{relabel}\ t :: \mathit{Labeled}\ H\ \tau$, where $t :: \mathit{Labeled}\ L\ \tau$. Suppose

$$\varepsilon_{\ell_A}(\mathit{relabel}\ t :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \mathit{relabel}_\bullet\ \varepsilon_{\ell_A}(t) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{relabel}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(t_1\ \langle * \rangle\ t_2 :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \varepsilon_{\ell_A}(t_1)\ \langle * \rangle_\bullet\ \varepsilon_{\ell_A}(t_2) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \varepsilon_{\ell_A}(t_1)\ \langle * \rangle\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{fmap}\ t_1\ t_2 :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \mathit{fmap}_\bullet\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{fmap}\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

(a) Erasure for flexible labeled values.

Terms: $t ::= \cdots \mid \mathit{relabel}_\bullet\ t \mid t_1\ \langle * \rangle_\bullet\ t_2 \mid \mathit{fmap}_\bullet\ t_1\ t_2$

(RELABEL$_{\bullet 1}$)
$$\frac{t \rightsquigarrow t'}{\mathit{relabel}_\bullet\ t \rightsquigarrow \mathit{relabel}_\bullet\ t'}$$

(RELABEL$_{\bullet 2}$)
$$\mathit{relabel}_\bullet\ (\mathit{Labeled}\ t) \rightsquigarrow \mathit{Labeled}\ \bullet$$

($\langle * \rangle_{\bullet 1}$)
$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ \langle * \rangle_\bullet\ t_2 \rightsquigarrow t_1'\ \langle * \rangle_\bullet\ t_2}$$

($\langle * \rangle_{\bullet 2}$)
$$\frac{t_2 \rightsquigarrow t_2'}{(\mathit{Labeled}\ t_1)\ \langle * \rangle_\bullet\ t_2 \rightsquigarrow (\mathit{Labeled}\ t_1)\ \langle * \rangle_\bullet\ t_2'}$$

($\langle * \rangle_{\bullet 3}$)
$$(\mathit{Labeled}\ t_1)\ \langle * \rangle_\bullet\ (\mathit{Labeled}\ t_2) \rightsquigarrow \mathit{Labeled}\ \bullet$$

(FMAP$_\bullet$)
$$\mathit{fmap}_\bullet\ t_1\ t_2 \rightsquigarrow (\mathit{Labeled}\ \bullet)\ \langle * \rangle_\bullet\ t_2$$

(b) Semantics of $\mathit{fmap}_\bullet$, $\langle * \rangle_\bullet$ and $\mathit{relabel}_\bullet$.

Fig. 27: Security of flexible labeled values.

we defined $\varepsilon_L(\mathit{relabel}\ t :: \mathit{Labeled}\ H\ \tau) = \mathit{relabel}\ \varepsilon'_L(t :: \mathit{Labeled}\ L\ \tau)$, for some suitable ε'_L that exhibits the desired behavior, e.g., $\varepsilon'_L(\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int}) = \mathit{Labeled}\ \bullet$. Even though this definition respects the simulation diagram of rule [RELABEL$_2$], introducing a *different* erasure function in a *context-sensitive* way is fatal for simulation of beta reductions. More precisely, the original erasure function is no longer *homomorphic over substitution*, i.e., $\varepsilon_{\ell_A}([t_1/x]\ t_2) \not\equiv [\varepsilon_{\ell_A}(t_1)/x]\ \varepsilon_{\ell_A}(t_2)$, a fundamental property for proving simulation of rule [BETA] [55,76,124,143,145]. As a counterexample, consider term $(\lambda x.\mathit{relabel}\ x)\ t$, which is erased *homomorphically*, that is $(\lambda x.\mathit{relabel}\ x)\ \varepsilon_L(t)$, and then beta-reduces along the orange path to $\mathit{relabel}\ \varepsilon_L(t)$. On the cyan path term $(\lambda x.\mathit{relabel}\ x)\ t$ beta-reduces to $\mathit{relabel}\ t$ and then is *context-sensitively*

erased to $relabel\ \varepsilon'_L(t)$. Observe that $relabel\ \varepsilon_L(t) \not\equiv relabel\ \varepsilon'_L(t)$, because we chose ε'_L so that it behaves differently from ε_L, specifically for public labeled values, i.e., when $t = Labeled\ 42 :: Labeled\ L\ Int$. Intuitively, function ε_{ℓ_A} should be oblivious to the context because terms can change context arbitrarily through beta reductions. To the best of our knowledge, this is the first work that points out this issue. This insight led us to uncover problems in the proof of single-step simulation in previous work on **LIO** [141, 145]. We refer the interested reader to Appendix A of [153] for details.

To recover single-step simulation for context-sensitive primitives such as $relabel$, we apply our *two-step erasure* technique and obtain an erasure function that is homomorphic over substitution *and* context-aware at the same time. Figure 27a defines the erasure function for $relabel$, which is replaced with $relabel_\bullet$, when it upgrades labeled data above the attacker's security level ($\ell_H \not\sqsubseteq \ell_A$). Then, primitive $relabel_\bullet$ simulates primitive $relabel$ for secret data by means of the rules in Figure 27b. In particular, rule [RELABEL$_1$] is simulated by rule [RELABEL$_{\bullet 1}$] and rule [RELABEL$_2$] by rule [RELABEL$_{\bullet 2}$], which simply produces $Labeled\ \bullet$. Even though secret data is actually erased in [RELABEL$_{\bullet 2}$], the simulation property still requires to apply erasure homomorphically to the argument of $relabel_\bullet$, so that the erasure function is homomorphic over substitution and rule [RELABEL$_{\bullet 1}$] simulates rule [RELABEL]. Intuitively, the erasure function should not remove public data until it gets upgraded to a secret security level, which happens exactly at the [RELABEL$_2$] ([RELABEL$_{\bullet 2}$]) reduction.

Similar issues arise in the simulation diagram of the applicative functor operator $\langle * \rangle$, which also requires a context-sensitive erasure function when applied to secret data. Consider the term $(Labeled\ t_1)\ \langle * \rangle\ (Labeled\ t_2) :: Labeled\ H\ Int$, which reduces to $Labeled\ (t_1\ t_2)$, according to rule [$\langle * \rangle_3$] from Figure 24. If we homomorphically apply the erasure function to this term, i.e., $\varepsilon_L(Labeled\ t_1)\ \langle * \rangle\ \varepsilon_L(Labeled\ t_2) = (Labeled\ \bullet)\ \langle * \rangle\ (Labeled\ \bullet)$, then the term reduces along the orange path to term $Labeled\ (\bullet\ \bullet) \not\equiv Labeled\ \bullet$, which we obtain instead, if we erase the term $Labeled\ (t_1\ t_2)$ along the cyan path. Observe that rule [$\langle * \rangle_3$] produces a function application within a $Labeled$ constructor, therefore it cannot possibly commute with erasure for secret labeled values, which always rewrites their content to $\bullet$.[26] To recover the single-step simulation property, we apply our *two-steps erasure* technique again. For secret labeled values, the erasure function replaces $\langle * \rangle$ with a new term $\langle * \rangle_\bullet$ (Figure 27a). Then, erasure is performed by means of rules [$\langle * \rangle_{\bullet 1}$,$\langle * \rangle_{\bullet 2}$, $\langle * \rangle_{\bullet 3}$] (Fig-

[26] In our conference version [152], rule [$\langle * \rangle_3$] raises a problem also for public labeled values, because the erasure function is not homomorphic over function application, in particular $\varepsilon_L(t_1\ \ t_2 :: MAC\ H\ \tau) = \bullet \not\equiv \varepsilon_L(t_1)\ \ \varepsilon_L(t_2)$. In that work, we replace function application with substitution in rule [$\langle * \rangle_3$], i.e. $(Labeled\ (\lambda x.t_1))\ \langle * \rangle\ (Labeled\ t_2) \rightsquigarrow Labeled\ (t_1\ [x\ /\ t_2])$, which solves the problem at the price of having a non-standard stricter semantics for $\langle * \rangle$. The erasure function presented here is homomorphic over function application and the semantics of $\langle * \rangle$ is standard.

ure 27b), which simulate rules $[\langle * \rangle_1, \langle * \rangle_2, \langle * \rangle_3]$ from Figure 24, respectively. In particular, to respect the single step-simulation property, rule $[\langle * \rangle_{\bullet 3}]$ ignores the content of the labeled values and simply reduces to $Labeled\ \bullet$. Since $fmap$ is defined in terms of $\langle * \rangle$, the erasure function replaces it with a new node $fmap_\bullet$ (when applied to secret data), which reduces to $\langle * \rangle_\bullet$, by means of rule [FMAP$_\bullet$], similar to rule [FMAP]. Terms $fmap_\bullet$, $\langle * \rangle_\bullet$ and $relabel_\bullet$ and their semantics rules are introduced in the calculus as a device to prove the single-step simulation property (they occur only in *erased* programs), just like nodes $join_\bullet$, $new_\bullet$, and $write_\bullet$.

Before proving noninterference for our concurrent model, we study the formal security requirements that concrete schedulers must satisfy to preserve security.

10.2 Scheduler Requirements

In this section, we take advantage of our general concurrent model featuring abstract scheduler state and semantics to prove a scheduler-parametric security condition. For this reason, we study what scheduling properties suffice to prove progress-sensitive noninterference in our model. Then, we show that the round-robin scheduler model (§8.3) satisfies these conditions and derive noninterference for our model of **MAC** by simply instantiating our main theorem. More precisely, the proofs hold for i) deterministic schedulers, that additionally ii) fulfill a restricted variant of single-step simulation (Figure 14), iii) do not leak secret information when scheduling sensitive threads, and iv) guarantee progress of observable threads i.e., schedulers may not allow secret threads to defer the execution of observable threads *indefinitely*.

We now formally characterize these scheduler requirements. In the following, labels ℓ_{L} and ℓ_{H} denote security levels that are respectively below and above the attacker's level, i.e., $\ell_{\mathrm{L}} \sqsubseteq \ell_A$ and $\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$.

Requirement 1

i) Determinancy:

$$\textit{If}\ \omega_1 \xrightarrow{(\ell,n,e)} \omega_2\ \textit{and}\ \omega_1 \xrightarrow{(\ell',n',e)} \omega_2',\ \textit{then}\ \ell \equiv \ell',\ n \equiv n'\ \textit{and}\ \omega_2 \equiv \omega_2'.$$

ii) Restricted simulation:

$$\textit{If}\ \omega_1 \xrightarrow{(\ell_L,n,e)} \omega_2,\ \textit{then}\ \varepsilon_{\ell_A}(\omega_1) \xrightarrow{(\ell_L,n,\varepsilon_{\ell_A}(e))} \varepsilon_{\ell_A}(\omega_2).$$

iii) No observable effect:

$$\textit{If}\ \omega_1 \xrightarrow{(\ell_H,n,e)} \omega_2,\ \textit{then}\ \omega_1 \approx_{\ell_A} \omega_2.$$

iv) Progress:

$$\textit{If}\ \omega_1 \xrightarrow{(\ell_L,n,e)} \omega_1'\ \textit{and}\ \omega_1 \approx_{\ell_A} \omega_2,\ \textit{then}\ \omega_2\ \textit{eventually schedules}\ (\ell_L, n).$$

Requirement i) (*determinancy*) ensures that the concurrent semantics is also deterministic and excludes refinement attacks [90]. Intuitively, Requirement ii) (*restricted simulation*) rules out schedulers that leverage sensitive information to make scheduling decisions about public threads.[27]. Requirement iii) (*no observable effect*) ensures that secret steps do not leak sensitive information to the scheduler state. The requirement expresses this formally by means of ℓ_A-equivalence for scheduler states, defined as $\omega_1 \approx_{\ell_A} \omega_2$ if and only if $\varepsilon_{\ell_A}(\omega_1) \equiv \varepsilon_{\ell_A}(\omega_2)$, where the erasure function of the scheduler state is scheduler specific and thus left unspecified. Requirement iv) (*progress*) avoids revealing secret data by observing the progress of public threads. Intuitively, a program could reveal sensitive information by forcing a secret thread to induce starvation of a public thread and thus suppressing a public event. This requirement excludes schedulers that enable starvation-based attacks, because it ensures that the same public thread will *eventually* get scheduled. The formal definition of "eventually" is technically interesting. Since we wish to make our proof modular, our model is parametric in the scheduler, which is considered in isolation from the thread pool. In this situation, we cannot predict how long the high threads are going to run, because the scheduler is decoupled from the thread pool. We overcome this technicality by indexing the ℓ_A-equivalence relation between scheduler states, which encode a single-step progress principle, i.e., Requirement 2 (explained below). Then, we use the indexes to exclude starvation by making the progress principle a well-founded induction principle, i.e., Requirement 3 (explained below).

Definition 2 (Annotated Scheduler ℓ_A-equivalence). *Two states are (i, j)-ℓ_A-equivalent, written $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$ if and only if $\omega_1 \approx_{\ell_A} \omega_2$ and i and j are upper bounds over the number of secret threads scheduled before the next common public thread in ω_1 and ω_2, respectively.*

The relation $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$ captures an alignment measure of two ℓ_A-equivalent states and how close they are to schedule the next common public thread. Informally, our noninterference proof excludes *starvation* of public threads by ensuring that two ℓ_A-equivalent schedulers will eventually align and schedule the same public thread, regardless of how the global configuration evolves. We capture the interplay between the (i, j)-ℓ_A-equivalent relationship and the evolution of schedulers in an unwinding-like condition [48]. Firstly, Requirement 2 ensures that if a scheduler runs a public thread, then any ℓ_A-equivalent scheduler can also make progress and run *some* thread, either a secret thread or the same public thread. Then, to eliminate starvation of the public thread,

[27] This condition is weaker from the corresponding requirement of the conference version of this paper [153], which additionally requires lock-step simulation for secret steps as well ($\ell_H \not\sqsubseteq \ell_A$). For secret steps, Requirement iii) (*no observable effect*) is sufficient. This weaker condition gives the same security guarantees and simplifies the formalization of a secure scheduler.

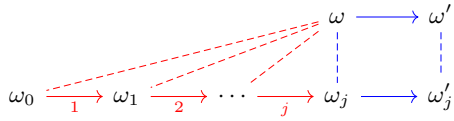


Fig. 28: Two (i,j)-ℓ_A-equivalent schedulers align in at most j steps.

Requirement 3 demands that the indexes of the (i,j)-ℓ_A-equivalence relation decrease strictly after every reduction.

Requirement 2 (Progress) *Given* $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_1'$, *and* $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$ *then:*

- *If* $j = 0$, *then* $\forall\ e'\ \exists\ \omega_2'$: $\omega_2 \xrightarrow{(\ell_L,n,e')} \omega_2'$.
- *If* $j > 0$, *then there exists* ℓ_H, n' *such that* $\forall\ e'\ \exists\ \omega_2'$: $\omega_2 \xrightarrow{(\ell_H,n',e')} \omega_2'$.

In the requirement, the (i,j)-ℓ_A-equivalence relation between the schedulers ensures that the second scheduler runs *at most* j secret threads before the same public thread. If the schedulers are aligned ($j = 0$), the second scheduler runs the public thread (ℓ_L, n) with any event e'.[28] If the schedulers are not aligned ($j > 0$), the second scheduler cannot predict what event the thread (ℓ_H, n') will trigger, therefore, as a conservative approximation, the step may involve *any* possible event e', which in turn determines the final state ω_2'. In principle, by repeatedly applying *progress* (Requirement 2), *no observable effect* (Requirement 1.iii) and transitivity of ℓ_A-equivalence, we could build a chain of secret steps that leads to the common public step and thus *eventually* align. Such a chain of steps is sketched in Figure 28, where the color of the scheduler steps denote running a secret (red for ℓ_H) or a public (blue for ℓ_L) thread and the dashed lines link ℓ_A-equivalent states. Unfortunately, the requirements and properties mentioned above are not enough. Intuitively, these propositions form a recursive argument, which is not well founded and allows secret threads to starve public threads, e.g., if scheduled *non-preemptively* [55]. The following requirement ensures that the chain of scheduler steps is finite and thus that public threads cannot starve indefinitely due to secret threads.

Requirement 3 (No Starvation) *Given* $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_1'$, $\omega_2 \xrightarrow{(\ell_H,n',e')} \omega_2'$, *such that* $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$, *then there exist* j' *such that* $j' < j$ *and* $\omega_1' \approx^{(i,j')}_{\ell_A} \omega_2'$.

[28] Requirement 1.ii) (*restricted simulation*) ensures that events e and e' are ℓ_A-equivalent, i.e., $e \approx_{\ell_A} e'$ iff $\varepsilon_{\ell_A}(e) \equiv \varepsilon_{\ell_A}(e')$. In our conference version [153], the requirement demands exactly the same event e, which is too strict for fork events. A public event $\mathit{Fork}\ \ell_H\ n$ contains secret information, namely the number n of secret threads, which might differ in the other run, e.g., $\mathit{Fork}\ \ell_H\ n' \not\equiv \mathit{Fork}\ \ell_H\ n$. The ℓ_A-equivalence relation for events handles this case as well: $\mathit{Fork}\ \ell_H\ n \approx_{\ell_A} \mathit{Fork}\ \ell_H\ n'$, because $\varepsilon_{\ell_A}(\mathit{Fork}\ \ell_H\ n) \equiv \varepsilon_{\ell_A}(\mathit{Fork}\ \ell_H\ n') \equiv \mathit{Step}$.

Now, using *progress* (Requirement 2), *no observable effect* (Requirement 1.iii), transitivity of ℓ_A-equivalence together with *no starvation* (Requirement 3), we can show that two ℓ_A-equivalent schedulers *eventually* align, see Figure 28.[29] Given two ℓ_A-equivalent scheduler states, i.e., $\omega \approx_{\ell_A}^{(i,j)} \omega_0$, such that $j > 0$, ω runs a public thread, i.e., $\omega \longrightarrow \omega'$, we apply *progress* (Requirement 2) and obtain a secret scheduler step from ω_0 to some scheduler state ω_1, i.e., $\omega_0 \longrightarrow \omega_1$. Then, we apply *no observable effect* (Requirement 1.iii) to the secret step $\omega_0 \longrightarrow \omega_1$ and derive ℓ_A-equivalence of these states, i.e., $\omega_0 \approx_{\ell_A} \omega_1$. By transitivity of the ℓ_A-equivalence relation for scheduler states, applied to $\omega \approx_{\ell_A} \omega_0$ and $\omega_0 \approx_{\ell_A} \omega_1$, we derive $\omega \approx_{\ell_A} \omega_1$. Then we lift ℓ_A-equivalence to annotated ℓ_A-equivalence, i.e., there exists some index j', such that $\omega \approx_{\ell_A}^{(i,j')} \omega_0$. Lastly, *no starvation* (Requirement 3) ensures that $j' < j$, which gives us a well-founded inductive principle. After repeating this process at most j times (j is strictly smaller after each step), we obtain $\omega_j \approx_{\ell_A}^{(i,0)} \omega$, we apply *progress* (Requirement 2) one last time. Then, since $j = 0$, the two schedulers are aligned and ω_j runs the public thread, stepping to ω'_j, i.e., $\omega_j \longrightarrow \omega'_j$. Finally, we combine *restricted simulation* (Requirement (i)) and scheduler *determinancy* (ii)) (Requirement i) and derive ℓ_A-equivalence of the final scheduler states, i.e., $\omega \approx_{\ell_A} \omega'_j$. The proof sketch above highlights the reasoning behind our proof technique—to avoid clutter, we have omitted details about scheduler events, which come from the concurrent semantics—we refer to our mechanized proof for more details. The requirements presented in this section capture a class of secure schedulers that are compatible with our concurrent calculus.

Definition 3 (Non-Interfering Scheduler).
A scheduler is non-interfering if it is satisfies requirements 1,2, and 3.

Before proving noninterference, we show that the round-robin scheduler is non-interfering.

Round Robin. We show that the round-robin scheduler from Section 8.3 fulfills all the requirements of a non-interfering scheduler and thus is a suitable candidate scheduler for our calculus. Firstly, we define erasure for the concrete scheduler state to simply remove secret threads from the queue, i.e., $\varepsilon_{\ell_A}(\omega) = \mathit{filter}\ (\lambda(\ell, n) \rightarrow \ell \sqsubseteq \ell_A)\ \omega$. Then, Figure 29 defines annotated ℓ_A-equivalence as an inductive relation. If both queues are empty, then no public thread is scheduled and rule [Empty] set both indexes to 0. If the next thread scheduled in the first or in the second queue is secret, rules [Secret$_1$] and [Secret$_2$] increase the corresponding index. Lastly, if the same public thread is scheduled *next* in both queues, rule [Public] sets both indexes to 0.

Proposition 4 *The round-robin scheduler is non-interfering.*

[29] In our conference version [153], *progress* (Requirement 2) and *no starvation* (Requirement 3) are combined, but technically we need to split these two requirements. Progress of the concurrent configuration requires no starvation, while no starvation ensures a well-founded inductive principle.

$$\text{(Empty)}\quad [] \approx_{\ell_A}^{(0,0)} []$$

$$\text{(Secret}_1\text{)}\quad \frac{\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2}{(\ell_H, n) : \omega_1 \approx_{\ell_A}^{(1+i,j)} \omega_2}$$

$$\text{(Secret}_2\text{)}\quad \frac{\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2}{\omega_1 \approx_{\ell_A}^{(i,1+j)} (\ell_H, n) : \omega_2}$$

$$\text{(Public)}\quad \frac{\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2}{(\ell_L, n) : \omega_1 \approx_{\ell_A}^{(0,0)} (\ell_L, n) : \omega_2}$$

Fig. 29: Annotated ℓ_A-equivalence (round robin).

Proof (Sketch). Firstly, the round-robin scheduler reductions in Figure 22 are clearly *deterministic* (Requirement i). Secondly, we show that the round-robin scheduler satisfies *restricted simulation* (Requirement ii) and *no observable effect*, (Requirement iii), by case analysis on the concurrent event and using simple properties over lists. Thirdly, we show *progress* (Requirement 2) by straightforward case split on the second index j, followed by case analysis on the scheduler step and the annotated ℓ_A-equivalence relation from Figure 29. Lastly, the round robin scheduler is *starvation-free* (Requirement 3), informally because it has a finite time-slot and is preemptive.

10.3 Progress-sensitive Non-interference

The proof of progress-sensitive noninterference relies on lemmas and propositions similar to the scheduler requirements from above. In the following, we write $c_1 \hookrightarrow_{(\ell,n)} c_2$ for a concurrent step $c_1 \hookrightarrow c_2$ that runs thread (ℓ, n). As before, labels ℓ_L and ℓ_H imply that $\ell_L \sqsubseteq \ell_A$ and $\ell_H \not\sqsubseteq \ell_A$, and notation $\hookrightarrow^\star$ denotes the reflexive transitive closure of the concurrent stepping relation $\hookrightarrow$. Finally, we extend ℓ_A equivalence to configurations, i.e., $c_1 \approx_{\ell_A} c_2$ if and only if $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$, and lift the indexes from the scheduler annotated ℓ_A-equivalence relation to it, i.e., if $c_1 = \langle \omega_1, \Sigma_1, \Phi_1 \rangle$, $c_2 = \langle \omega_2, \Sigma_2, \Phi_2 \rangle$, then $c_1 \approx_{\ell_A}^{(i,j)} c_2$ if and only if $c_1 \approx_{\ell_A} c_2$ and $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$.

Proposition 5

i) Determinancy*: if* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$ *then* $c_2 \equiv c_3$.
ii) Restricted simulation*: if* $c_1 \hookrightarrow_{(\ell_L,n)} c_2$ *then* $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,n)} \varepsilon_{\ell_A}(c_2)$.
iii) No observable effect*: if* $c_1 \hookrightarrow_{(\ell_H,n)} c_2$ *then* $c_1 \approx_{\ell_A} c_2$.

Using Proposition 5, we show that the concurrent semantics preserves ℓ_A-equivalence.

Proposition 6 ($\approx_{\ell_A}$ Preservation) *If* $c_1 \approx_{\ell_A} c_2$ *and* $c_1 \hookrightarrow_{(\ell,n)} c_1'$*, then*

- *If* $\ell \not\sqsubseteq \ell_A$*, then* $c_1' \approx_{\ell_A} c_2$.
- *If* $\ell \sqsubseteq \ell_A$ *and* $c_2 \hookrightarrow_{(\ell,n)} c_2'$*, then* $c_1' \approx_{\ell_A} c_2'$.

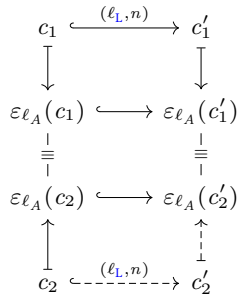


Fig. 30: 1-Step Progress.

The theorem of *progress-sensitive* noninterference requires to prove that two ℓ_A-equivalent configurations remain related, even if only one configuration steps. When the step involves a secret thread, the theorem follows easily by preservation of ℓ_A-equivalence (Proposition 6) and transitivity of the ℓ_A-equivalence relation. The challenging part of the proof consists in showing *progress* of a public thread, which requires reconstructing the execution of a certain number of secret threads until the same public thread takes over, similar to the diagram for scheduler progress in Figure 28. The proof of *progress* consists of two parts. The first part reconstructs the execution of the secret threads that precede the public thread in the schedule (*scheduler progress*). In this part, the reconstruction of the secret steps is facilitated by the fact that the concurrent semantics always steps and that *no observable event* (Proposition 5.iii) ensures that secret steps preserve ℓ_A-equivalence. Then, in the second step, *restricted simulation* (Proposition 5.ii) simulates the execution of the public thread under erasure. Lastly, the corresponding public step in the second configuration has to be reconstructed using the erased step and some *valid* side-conditions that we proceed to formalize (we motivate the need for these standard assumptions below).

Definition 4 (Valid Configuration). *A concurrent configuration is valid if and only if contains valid memory references, and it does not contain terms* $\bullet$, $new_\bullet$, $write_\bullet$, $fork_\bullet$, $fmap_\bullet$, $\langle * \rangle_\bullet$, *and* $relabel_\bullet$.

Assuming valid configurations, we can prove *1-step progress*, i.e., the reconstruction of the public step.

Proposition 7 (1-Step Progress) *If* $c_1 \approx^{(i,0)}_{\ell_A} c_2$, $c_1 \hookrightarrow_{(\ell_L,n)} c_1'$ *and* c_2 *is valid, then there exists* c_2' *such that* $c_2 \hookrightarrow_{(\ell_L,n)} c_2'$.

Proof (Sketch). The diagram in Figure 30 gives an overview of this proposition, which requires reconstructing the dashed horizontal line. Since $c_1 \approx^{(i,0)}_{\ell_A} c_2$ and the scheduler in c_1 runs a public thread, then the scheduler in c_2 is *aligned* (the second index in the ℓ_A-equivalence relation is 0) and thus runs the same

thread by *scheduler progress* (Proposition 2). Then, *restricted simulation* (Proposition 5.ii) applied to step $c_1 \hookrightarrow_{(\ell_L,n)} c_1'$ gives the erased step $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,m)} \varepsilon_{\ell_A}(c_1')$. Lastly, using the erased step, the assumption that c_1 and c_2 are *valid* (Definition 4), and ℓ_A-equivalent, i.e., $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$, it is possible to reconstruct c_2' together with step $c_2 \hookrightarrow_{(\ell_L,n)} c_2'$ (the dashed horizontal path in the diagram).

Validity. The following example motivates why the reconstruction of the public step in *1-step progress* (Proposition 7) requires the extra assumptions about *valid* configurations. Informally, the fact that public threads can write to secret resources (e.g., memories and thread pools) complicates the reconstruction of the step: the erasure function removes these resources in the erased step. For example, the erasure function replaces secret memories and addresses with $\bullet$, which makes it impossible to replay secret write operations. Concretely, consider a step in which a public thread writes to a secret memory, e.g., $\langle \Sigma, \mathit{write}\ (\mathit{Ref}\ n)\ t\rangle \longrightarrow \langle \Sigma(\ell_H)[n \mapsto t], \mathit{return}\ ()\rangle$. A low-equivalent program could be $\langle \Sigma', \mathit{write}\ (\mathit{Ref}\ n')\ t'\rangle$, for some store Σ', address n' and term t' such that $\Sigma \approx_{\ell_A} \Sigma'$, $\mathit{Ref}\ n \approx_{\ell_A} \mathit{Ref}\ n'$ and $t \approx_{\ell_A} t'$. Unfortunately, there is no guarantee that address n' points within the bounds of the secret memory $\Sigma'(\ell_H)$. In particular, the erasure function maps *both* valid and invalid references to $\mathit{Ref}\ \bullet$, therefore knowing that n points within the bounds of $\Sigma(\ell_H)$ does not guarantee that n' is valid for $\Sigma'(\ell_H)$. Since term reduction gets stuck for out-of-bounds memory operations, the reconstruction of this step is impossible, unless the address n' is known to be valid. The assumptions about valid configuration serve to solve exactly these technicalities. We conclude this remark with the proof that validity is an invariant of the semantics of the calculus.

Proposition 8 (Valid Invariant) *If c_1 is valid and $c_1 \hookrightarrow c_2$ then c_2 is valid.*

Using the invariant, we now extend *1-step progress* to *multi-step* progress.

Proposition 9 (Progress) *If $c_1 \approx_{\ell_A}^{(i,j)} c_2$, $c_1 \hookrightarrow_{(\ell_L,n)} c_1'$, and c_1, c_2 are valid configurations, then there exists c_2' and c_2'' such that $c_2 \hookrightarrow^\star c_2' \hookrightarrow_{(\ell_L,n)} c_2''$.*

Proof (Sketch). *Scheduler progress* (Requirement 2) determines which is the next thread in the schedule and drives the proof by case analysis on j:

- If $j = 0$, then the scheduler in c_2 runs the same public thread (Requirement 2) and the proof follows from *1-step progress* (Proposition 7);
- If $j > 0$, then the scheduler runs a secret thread (Requirement 2), which reduces to some intermediate configuration c_2', i.e., $c_2 \hookrightarrow_{(\ell_H,n')} c_2'$ where $c_2 \approx_{\ell_A} c_2'$ by *no observable effect* (Proposition 5.iii). Then, by transitivity of the ℓ_A-equivalence relation for configurations applied to $c_1 \approx_{\ell_A} c_2$ and $c_2 \approx_{\ell_A} c_2'$, we obtain $c_1 \approx_{\ell_A} c_2'$ and lift the relation to indexed ℓ_A-equivalence, i.e., $c_1 \approx_{\ell_A}^{(i,j')} c_2$ for some index j'. Lastly, *no starvation* (Requirement 3) ensures that $j' < j$ and the proof follows by well-founded induction.

Finally, we prove progress-sensitive noninterference by combining *progress*, (Proposition 9) and ℓ_A-equivalence *preservation* (Proposition 6).

Theorem 2 (Progress-sensitive noninterference) *Given valid global configurations c_1, c_1', c_2, and a non-interfering scheduler, if $c_1 \approx_{\ell_A} c_2$ and $c_1 \hookrightarrow c_1'$, then there exists c_2' such that $c_2 \hookrightarrow^\star c_2'$ and $c_2 \approx_{\ell_A} c_2'$.*

To conclude, we instantiate scheduler-parametric progress-sensitive noninterference (Theorem 2) with the proof that the round-robin scheduler is non-interfering (Proposition 4) and derive security for our model of **MAC**.

Corollary 1 ***MAC** satisfies progress-sensitive noninterference.*

11 Related work

Mechanized Proofs. Russo presents the library **MAC** as a functional pearl and relies on its simplicity to convince readers about its correctness [123]. This work bridges the gap on **MAC**'s lack of formal guarantees and exhibits interesting insights on the proofs of its soundness. **LIO** is a library structurally similar to **MAC** but dynamically enforcing IFC [145]. The core calculus of **LIO**, i.e., side-effect free computations together with exception handling, has been modeled in the Coq proof assistant [143]. Different from our work, these mechanized proofs do not simplify the treatment of sensitive exceptions by masking them in erased programs. In parallel to [143], Breeze [59] is a pure programming language that explores the design space of IFC and exceptions, which is accompanied with mechanized proofs in Coq. Bichhawat et al. develop an intra-procedural analysis for Javascript bytecode, which prevents implict leaks in presence of exceptions and unstructured control flow constructs [21].

Parametricity. Parametric polymorphism prevents a polymorphic function from inspecting its argument. In a similar manner, a non-interferent program cannot change its observable behaviour depending on the secret. Researchers have explored further this deep and subtle connection by obtaining a translation from DCC [1] to System F in order to leverage on parametricity [150]. Shikuma and Igarashi [136] points out an error on such translation and gives a counterexample of a leaked translation. Recently, Bowman and Ahmed [25] provide a sound translation from DCC into System F.

Concurrency. Considering IFC for a general scheduler could lead to refinements attacks. In this light, Russo and Sabelfeld provide termination-insensitive noninterference for a wide-class of deterministic schedulers [126]. Barthe et al. adopt this idea for Java-like bytecode [17]. Although we also consider deterministic schedulers, our security guarantees are stronger by considering leaks of information via abnormal termination. Heule et al. describe how to retrofit IFC in a programming language with sandboxes [55]. Similar to our work, their soundness proofs are parametric on deterministic schedulers and provide progress-sensitive noninterference with informal arguments regarding

thread progress—in this work, we spell out formal requirements on schedulers capable to guarantee thread progress. A series of work for π-calculus consider non-deterministic schedulers while providing progress-sensitive noninterference [57, 58, 66, 115]. Mantel and Sudbrock propose a novel scheduler-independent trace-based information-flow control property for multi-threaded programs and identify the class of *robust scheduler*, which satisfy that condition [85]. While there are some similarities between the requirements of those robust schedulers and those discussed here in Section 10.2, that work assumes terminating threads, while our progress-sensitive noninterference theorem does not.

Security Libraries. Li and Zdancewic's seminal work [75] shows how the structure *arrows* can provide IFC as a library in Haskell. Tsai et al. extend that work to support concurrency and data with heterogeneous labels [149]. Russo et al. implement the security library **SecLib** using a simpler structure than arrows [124], i.e. monads—rather than labeled values, this work introduces a monad which statically label side-effect free values. The security library **LIO** [141, 145] enforces IFC for both sequential and concurrent settings dynamically. **LIO** presents operations similar to $fmap$ and $\langle * \rangle$ for labeled values with differences in the returning type due to **LIO**'s checks for clearence—this work provides a foundation to analyze the security implications of such primitives. Mechanized proofs for **LIO** are given only for its core sequential calculus [145]. Inspired by **SecLib** and **LIO**'s designs, **MAC** leverages Haskell's type system to enforce IFC [123] statically. Unlike **LIO**, data-dependent security policies cannot be expressed in **MAC**, due to its static nature. This limitation is addressed by **HLIO**, which provides a hybrid approach by means of some advanced Haskell's type-system features: IFC is statically enforced while allowing the programmers to defer selected security checks until run-time [30]. Several works have also investigated the use of dependent types to precisely capture the nature of data-dependent security policies [82, 94, 97, 102].

Our work studies the security implications of extending **LIO**, **MAC**, and **HLIO** with a rich structure for labeled values. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC in Haskell [38]. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [63]—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. Rather than running multiple copies of a program, Schmitz et al. provide a library with *faceted values* [132], where values present different behavior according to the privilege of the observer. Different from the work above, we present a fully-fledged mechanized proof for our sequential and concurrent calculus which includes references, synchronization variables, and exceptions.

IFC Tools. IFC research has produced compilers capable of preserving confidentiality of data: Jif [100] and Paragon [28] (based on Java), and flowcaml [137] (based on Caml). The SPARK language presents a IFC analysis which has been

extended to guarantee progress-sensitive non-inference [117]. JSFlow [51] is one of the state-of-the-art IFC system for the web (based on JavaScript). These tools preserve confidentiality in a fine-grained fashion where every piece of data is explicitly label. Specifically, there is no abstract data type to label data, so our results cannot directly apply to them.

Operating Systems. **MAC** borrows ideas from Mandatory Access Control (MAC) [19,20] and phrases them into a programming language setting. Although originated in the 70s, there are modern manifestations of this idea [69, 96, 166], applied to diverse scenarios, like the web [18,146] and mobile devices [29,64]. Due to its complexity, it is not surprising that OS-based MAC systems lack accompanying soundness guarantees or mechanized proofs—**seL4** being the exception [96]. The level of abstractions handled by **MAC** and OSes are quite different, thus making uncertain how our insights could help to formalize OS-based MAC systems. MAC systems [19] assign a label with an entire OS process—settling a single policy for all the data handled by it. In principle, it would be possible to extend such MAC-like systems to include a notion of labeled values with the functor structure as well as the relabeling primitive proposed by this work. For instance, COWL [146] presents the notion of *labeled blob* and *labeled XHR* which is isomorphic to the notion of labeled values, thus making possible to apply our results. Furthermore, because many MAC-like system often ignore termination leaks [39, 166], there is no need to use call-by-name evaluation to obtain security guarantees.

12 Conclusion

We have presented the first *fully-fledged* formalization of **MAC** and proved that our model satisfies progress-insensitive noninterference through term erasure. Using our scheduler-parametric model, we have characterized a large class of deterministic schedulers which preserve security—something that has been only treated informally before [55, 141]. In particular, we present a scheduler-parametric noninterference proof and establish security for **MAC**'s model by simply instantiating our main theorem with the round-robin scheduler. To the best of our knowledge, this is the first work of its kind for IFC libraries in Haskell, both for completeness and number of features included in the model.

This work extends **MAC** with a *flexible labeled data* API, which gives a functor algebraic structure to sensitive data, in a way that it can be processed through classic functional programming patterns that do not incur in security checks. We support our results with 4,000 LOC of machine-checked proof scripts, that we have developed in the Agda proof assistant and have made available online. This effort led us to identify challenges in the security proofs involving context-sensitive security primitives and devise the *two-steps erasure* technique to address them. We hope that the insights of this work will be helpful in the verification of future IFC libraries and programming languages.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM.
2. Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.
3. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.
4. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, March 2005.
5. Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *Special issue of ACM Transactions on Information and System Security (TISSEC)*, 2009.
6. Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Annual Network & Distributed System Security Symposium*. Internet Society, 2015.
7. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
8. K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.
9. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 159–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
10. Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*. ACM, 2007.
11. William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 101–113, 2015.
12. Niklas Broberg, Bart Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 217–232, Berlin, Heidelberg, 2013. Springer-Verlag.
13. Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Conference on Security*, SEC. USENIX Association, 2013.
14. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.
15. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proceedings of the 18th Nordic Conference on Secure IT Systems - Volume 8208*, NordSec 2013, pages 116–122, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

16. Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.
17. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communication of the ACM*, 20(7):504–513, July 1977.
18. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.
19. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *ACM Symposium on Operating Systems Principles*, SOSP. ACM, 2005.
20. Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.
21. J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.
22. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy*, pages 75–75, April 1984.
23. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing*. ACM, 2014.
24. Daniel Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.
25. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 11–31, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
26. Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure Information Flow as Typed Process Behaviour. In *European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.
27. Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 81–92, New York, NY, USA, 2002. ACM.
28. C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
29. J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.
30. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, pages 170–178, Berlin, Heidelberg, 2012. Springer-Verlag.
31. Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android. In *Computer Security – ESORICS 2013*, pages 775–792, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
32. Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4):291–347, December 2005.

33. Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP. ACM, 2007.
34. Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 16–, Washington, DC, USA, 2006. IEEE Computer Society.
35. Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, April 2010.
36. Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328, New York, NY, USA, 2015. ACM.
37. Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European Conference on Research in Computer Security*, ESORICS'10, pages 116–133, Berlin, Heidelberg, 2010. Springer-Verlag.
38. Simon Marlow. *Parallel and concurrent programming in Haskell*. O'Reilly, July 2013.
39. Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.
40. D. McCullough. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy(SP)*, volume 00, page 161, April 1987.
41. Simon Meurer and Roland Wismüller. Apefs: An infrastructure for permission-based filtering of android apps. In AndreasU. Schmidt, Giovanni Russello, Ioannis Krontiris, and Shiguo Lian, editors, *Security and Privacy in Mobile Information and Communication Systems*, volume 107. Springer Berlin Heidelberg, 2012.
42. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
43. Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.
44. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. *2012 IEEE Symposium on Security and Privacy*, 0, 2013.
45. Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE Computer Security Foundations Symposium*, pages 417–431, Lisbon, Portugal, June 2016.
46. Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.
47. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.
48. Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, SP. IEEE Computer Society, 2011.
49. Mathias V. Pedersen and Aslan Askarov. From trash to treasure: Timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 693–709, 2017.

50. François Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.
51. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.
52. Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. Progress-sensitive security for spark. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, pages 20–37, Berlin, Heidelberg, 2016. Springer-Verlag.
53. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2009.
54. Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 280–288, New York, NY, USA, 2015. ACM.
55. Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 13–24, New York, NY, USA, 2008. ACM.
56. Alejandro Russo and Andrei Sabelfeld. Security for multithreaded programs under cooperative scheduling. In Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, pages 474–480, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
57. Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *The Journal of Logic and Algebraic Programming*, 78(7):593 – 618, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007).
58. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.
59. Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca Viganò, editors, *POST*, volume 9635 of *LNCS*. Springer, 2016.
60. Naokata Shikuma and Atsushi Igarashi. Proving noninterference by a fully complete translation to the simply typed λ-calculus. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06, pages 301–315, Berlin, Heidelberg, 2007. Springer-Verlag.
61. V. Simonet. The Flow Caml system. Software release at http://cristal.inria.fr/ simonet/soft/flowcaml/, 2003.
62. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.
63. Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 718–735, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
64. Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International*

Conference on Functional Programming, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.

65. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457, to appear in Journal of Functional Programming*, 2012.
66. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.
67. Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014.
68. Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 187–202, Washington, DC, USA, 2007. IEEE Computer Society.
69. Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 115–125, New York, NY, USA, 2004. ACM.
70. Marco Vassena, Joachim Breitner, and Alejandro Russo. Securing concurrent lazy programs against information leakage. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 37–52, 2017.
71. Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 538–557, 2016.
72. Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 15–28, New York, NY, USA, 2016. ACM.
73. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.
74. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.
75. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 563–574, New York, NY, USA, 2011. ACM.
76. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *ACM Conference on Programming Language Design and Implementation*. ACM, 2012.

$$(\textsc{Relabel}_\chi)\quad relabel\ (Labeled_\chi\ t) \rightsquigarrow Labeled_\chi\ t$$

$$(\textsc{Relabel}_{\bullet\chi})\quad relabel_\bullet\ (Labeled_\chi\ t) \rightsquigarrow Labeled\ \bullet$$

$$(\langle * \rangle_{\chi 1})\quad \frac{t_2 \rightsquigarrow t_2'}{(Labeled_\chi\ t_1)\ \langle * \rangle\ t_2 \rightsquigarrow (Labeled_\chi\ t_1)\ \langle * \rangle\ t_2'}$$

$$(\langle * \rangle_{\chi 2})\quad (Labeled_\chi\ t_1)\ \langle * \rangle\ (Labeled\ t_2) \rightsquigarrow Labeled_\chi\ t_1$$

$$(\langle * \rangle_{\chi 3})\quad (Labeled_\chi\ t_1)\ \langle * \rangle\ (Labeled_\chi\ t_2) \rightsquigarrow Labeled_\chi\ t_1$$

$$(\langle * \rangle_{\chi 4})\quad (Labeled\ t_1)\ \langle * \rangle\ (Labeled_\chi\ t_2) \rightsquigarrow Labeled_\chi\ t_2$$

Fig. 31: Semantics of flexible labeled values with exceptions.

Appendix

A Flexible Labeled Values in Sequential MAC

This section adjusts the semantics of flexible labeled values described in Section 9 for the sequential setting, in which labeled values have an additional constructor for exceptional labeled values. This constructor ($Labeled_\chi$) masks sensitive exceptions raised in a secret context embedded through *join* from leaking into an outer public context. Figure 31 shows the new reduction rules for sequential primitives *relabel* and $\langle * \rangle$, which simply propagate exceptional values by means of rules [$\textsc{Relabel}_\chi$, $\langle * \rangle_{\chi 1}$, $\langle * \rangle_{\chi 2}$, $\langle * \rangle_{\chi 3}$, $\langle * \rangle_{\chi 4}$]. Rule [$\textsc{Relabel}_{\bullet\chi}$] simulates rule [$\textsc{Relabel}_\chi$] when *relabel* upgrades a public exceptional labeled value to a label above the attacker's level. In order to *mask* the sensitive exception, the rule produces a non-exceptional value, i.e., $Labeled\ \bullet$. Rules [$\langle * \rangle_{\chi 1}$, $\langle * \rangle_{\chi 2}$, $\langle * \rangle_{\chi 3}$] are somewhat unusual: even though they propagate the first exception observed during evaluation, they still evaluate the second argument. For example, rule [$\langle * \rangle_{\chi 1}$] reduces the second argument unnecessarily—it is going to return the exception from the first argument anyway in rules [$\langle * \rangle_{\chi 2}$, $\langle * \rangle_{\chi 3}$]. Since the calculus is *non-strict*, it would have been more natural to replace all these rules (i.e., [$\langle * \rangle_{\chi 1}$, $\langle * \rangle_{\chi 2}$, $\langle * \rangle_{\chi 3}$]) with a more general rule [$\langle * \rangle_{\chi 123}$], which avoids evaluating the second argument when the first is exceptional, i.e., $Labeled_\chi\ t_1\ \langle * \rangle\ t_2 \rightsquigarrow Labeled_\chi\ t_1$. The semantics of $\langle * \rangle$ in Figure 31 appears unnecessarily *strict* in the second argument. Intuitively, the strict rules are equivalent to the non-strict rules, except for *non-terminating* terms. For example, consider the program $(Labeled_\chi\ t_1)\ \langle * \rangle\ \bot$, where $\bot$ indicates a non-terminating term. With the *strict* semantics, the program fails to terminate—it loops due to rule [$\langle * \rangle_{\chi 1}$] and thus it is indistinguishable from $\bot$. Instead,

with the *non-strict* semantics, the program terminates via rule $[\langle * \rangle_{\chi 123}]$, i.e., $(Labeled_\chi\ t_1)\ \langle * \rangle\ \bot \rightsquigarrow (Labeled_\chi\ t_1)$. In the sequential calculus, the semantics of the applicative functor operator $\langle * \rangle$ is strict in the second term, because this choice simplifies the security proofs. More precisely, the non-strict semantics of the alternative rule $[\langle * \rangle_{\chi 123}]$ does not respect the simulation diagram from Figure 14 for secret exceptional labeled values. Remember that the erasure function replaces sensitive exceptions with non-exceptional values, i.e., $\varepsilon_L(Labeled_\chi\ t_1 :: Labeled\ H\ \tau) = Labeled\ \bullet$. Then, the simulation diagram for rule $[\langle * \rangle_{\chi 123}]$, i.e., $Labeled_\chi\ t_1\ \langle * \rangle\ t_2 \rightsquigarrow Labeled_\chi\ t_1$ requires producing a step $(Labeled\ \bullet)\ \langle * \rangle_\bullet\ \varepsilon_L(t_2) \rightsquigarrow Labeled\ \bullet$, obtained from erasing the result, i.e., $\varepsilon_L(Labeled_\chi\ t_2 :: Labeled\ H\ \tau) = Labeled\ \bullet$. Unfortunately, none of the reduction rules of $\langle * \rangle_\bullet$ (i.e., rules $[\langle * \rangle_{\bullet 1}, \langle * \rangle_{\bullet 2}, \langle * \rangle_{\bullet 3}]$ from Figure 27b) can reproduce that step. (Notice that rule $[\langle * \rangle_{\bullet 3}]$ works only if the second argument is a labeled value, but not in the general case.) Intuitively, when using the non-strict version of the applicative functor operator, the erasure function should not mask sensitive exceptions: rule $[\langle * \rangle_{\chi 123}]$ is *sensitive* to exceptions! To recover the simulation property for that rule, the erasure function should be changed so that sensitive exceptions get preserved, i.e., $\varepsilon_L(Labeled_\chi\ t :: Labeled\ H\ \tau) = Labeled_\chi\ \bullet$. However, this definition breaks the simulation diagram of rules [JOIN,JOIN$_\chi$]. Then, since the strict and non-strict semantics are equivalent for *terminating* programs, the calculus adopts the strict variant. Notice that this choice does not weaken the security condition: the sequential calculus is already *termination insensitive* due to $join$.

B Thread Synchronization

This section extends **MAC** with thread synchronization primitives, which allows threads to coordinate in concurrent programs. Using synchronized mutable variables ($MVar$), **MAC** provide simple thread communication mechanisms such as binary semaphores and message queues. However, thread synchronization may also introduce covert channels: synchronous operations can suppress (e.g., due to a deadlock) or delay public side-effects and thus leak information. In order to avoid leaking information, **MAC** provides the following API for thread synchronization primitives.

$$
\begin{aligned}
&\textbf{data}\ MVar\ \ell\ \tau \\
&newMVar :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow MAC\ \ell_{\mathrm{L}}\ (MVar\ \ell_{\mathrm{H}}\ \tau) \\
&takeMVar :: MVar\ \ell\ \tau \rightarrow MAC\ \ell\ \tau \\
&putMVar :: MVar\ \ell\ \tau \rightarrow \tau \rightarrow MAC\ \ell\ ()
\end{aligned}
$$

To track information flows and enforce security, **MAC** associates labels with mutable synchronization variables, i.e., abstract data type $MVar\ \ell\ \tau$. These variables are either empty or full with a term of type τ at security level ℓ. Function $newMVar$ creates an *empty* synchronization variable at the same security level of the current thread, or above. Similarly to primitive new over references,

creating a synchronization variable constitutes a write operation, subject to the *no write-down* security policy. With function $takeMVar$, a thread reads synchronously from a a variable. If the variable is empty, the thread blocks, otherwise it returns the content and leaves the variable empty. Conversely, with function $putMVar$, a thread writes synchronously to a variable. If the variable is full, the thread blocks, otherwise it fills it with the argument. Observe the type signature of $takeMVar$ and $putMVar$. In contrast to the operations that read and write plain references (Section 6, primitives $takeMVar$ and $putMVar$ work within one security level. Intuitively, whenever a thread executes any of these operations is performing *both* read and write side-effects at the same time. For example, operation $putMVar$ writes to a variable only after reading the empty state and conversely for $takeMVar$. Then, the *no read-up* and *no write-down* security policies guarantee that these operations are secure only when the thread and the variable are at the same security level. The following example motivates this design decision. Consider a thread and a synchronization variables at arbitrary security levels ℓ_1 and ℓ_2, respectively. Primitive $putMVar$, allows the thread labeled with ℓ_1 to write to the variable at security level ℓ_2. Then, the no write-down policy demands that that the variable is at least as sensitive as the thread, i.e., $\ell_1 \sqsubseteq \ell_2$. However, primitive $putMVar$ also reads the state of the variable (it has to block the thread if the variable is already full). Then, the no read-up policy demands that the variable is no more sensitive than the thread, i.e., $\ell_2 \sqsubseteq \ell_1$. As a result, it follows that primitive $putMVar$ is secure only when the both constraints (i.e., $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$) are satisfied, i.e., when $\ell_1 \equiv \ell_2$ and the thread and the variable are at the same security level. Using the same line of though, we argue for security of the API of primitive $takeMVar$.

To formally prove security for these operations, we add synchronization variables and primitives to the concurrent calculus.

B.1 Semantics

To support thread synchronization primitives, Figure 32 adjusts the memory model to work with synchronization variables. A memory segment contains *cells*, which can be either empty, i.e., $\otimes$, or full with a term, i.e., $[\![t]\!]$.[30] A synchronization variable is represented as a value $MVar\ n :: MVar\ \ell\ \tau$ where n is an address, pointing to the n-th *cell* of the ℓ-memory, containing a slot for a term of type τ.[31] Rule [NewMVar] creates an empty memory cell in the ℓ-labeled memory, i.e., $\Sigma(\ell)[n \mapsto \otimes]$, at fresh address $n = |\Sigma(\ell)|$ and returns the variable $MVar\ n$. Rule [PutMVar$_1$] evaluates the variable and rule [PutMVar$_2$] fills the empty cell with the term, i.e., $\Sigma(\ell)[n \mapsto [\![t]\!]]$, and returns unit. The

[30] This memory model is more general than the model used for mutable references, which are then encoded as synchronization variables that are always full.

[31] **MAC** implements labeled synchronization variables ($MVar$) as a simple wrapper around the unlabeled synchronization variable from the standard library. For simplicity. we represent synchronization variables as a plain index, just like we did for plain references.

Memory ℓ: $t_s ::= [\,] \mid c : t_s$
Cells: $c ::= \otimes \mid [\![t]\!]$
Types: $\tau ::= \cdots \mid MVar\ \ell\ \tau$
Values: $v ::= \cdots \mid MVar\ n$
Terms: $t ::= \cdots \mid newMVar \mid takeMVar\ t \mid putMVar\ t_1\ t_2$

(NewMvar)
$$\frac{|\Sigma(\ell)| = n}{\langle \Sigma, newMVar\rangle \longrightarrow \langle \Sigma(\ell)[n \mapsto \otimes], return\ (MVar\ n)\rangle}$$

(PutMVar$_1$)
$$\frac{t_1 \rightsquigarrow t_1'}{\langle \Sigma, putMVar\ t_1\ t_2\rangle \longrightarrow \langle \Sigma, putMVar\ t_1'\ t_2\rangle}$$

(PutMVar$_2$)
$$\frac{\Sigma(\ell)[n] \equiv \otimes}{\langle \Sigma, putMVar\ (MVar\ n)\ t\rangle \longrightarrow \langle \Sigma(\ell)[n \mapsto [\![t]\!]], return\ ()\rangle}$$

(TakeMVar$_1$)
$$\frac{t \rightsquigarrow t'}{\langle \Sigma, takeMVar\ t\rangle \longrightarrow \langle \Sigma, takeMVar\ t'\rangle}$$

(TakeMVar$_2$)
$$\frac{\Sigma(\ell)[n] \equiv [\![t]\!]}{\langle \Sigma, takeMVar\ (MVar\ n)\rangle \longrightarrow \langle \Sigma(\ell)[n \mapsto \otimes], return\ t\rangle}$$

Fig. 32: **MAC** with synchronization primitives.

premise of rule [PutMVar$_2$] determines the *blocking* behavior of $putMVar$. For example, a thread that attempts to execute $putMVar$ on a full cell cannot reduce via [PutMVar$_2$] (it does not satisfy the premise $\Sigma(\ell)[n] \equiv \otimes$). Since the thread cannot execute through any other rule, it gets *stuck*. Rule [TakeMVar$_1$] evaluates the reference and rule [TakeMVar$_2$] returns the content of the corresponding non-empty cell, i.e., $\Sigma(\ell)[n] = [\![t]\!]$ for some term t, and empties it, i.e., $\Sigma(\ell)[n \mapsto \otimes]$. If the cell is empty, then the thread gets stuck and blocks, as explained above.

B.2 Erasure Function

Proving security for the synchronization primitives presented above is straightforward. The primitives are clearly *deterministic* and showing *single-step simulation* is even simpler than for references because primitives $putMVar$ and $takeMVar$ work within the same security level. The erasure function is homomorphic over memory cells (Figure 33a), rewrites the address of a secret reference to $\bullet$, and replaces term $newMVar$ with $newMVar_\bullet$, when it creates a sensitive synchronization variable (Figure 33b). Figure 33c shows the only

$$\varepsilon_{\ell_A}(\otimes) = \otimes \qquad \varepsilon_{\ell_A}([\![t]\!]) = [\![\varepsilon_{\ell_A}(t)]\!]$$

(a) Erasure for memory cells.

$$\varepsilon_{\ell_A}(newMVar :: MAC\ \ell_{\mathrm{L}}\ (MVar\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} newMVar_\bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ newMVar & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(MVar\ n :: MVar\ \ell_{\mathrm{H}}\ \tau) = \begin{cases} MVar\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ MVar\ n & \text{otherwise} \end{cases}$$

(b) Erasure for $newMVar$ and $MVar$.

Terms: $t ::= \cdots \mid newMVar_\bullet$

(NewMvar$_\bullet$)
$\langle \Sigma, newMVar_\bullet \rangle \longrightarrow \langle \Sigma, return\ (MVar\ \bullet) \rangle$

(c) Semantics of $newMVar_\bullet$.

Fig. 33: Erasure function for memory cells and synchronization primitives.

reduction rule [NewMVar$_\bullet$] for $newMVar_\bullet$, which simply returns a dummy reference, i.e., $MVar\ \bullet$, and leaves the store Σ unchanged. We do not need to apply two-steps erasure for primitives $putMVar$ and $takeMVar$, because a thread can only use them to read/write to a memory cell at the same security level. Thus either both the thread and the variable are secret and the whole computation collapses to $\bullet$, or both are public and erased homomorphically.

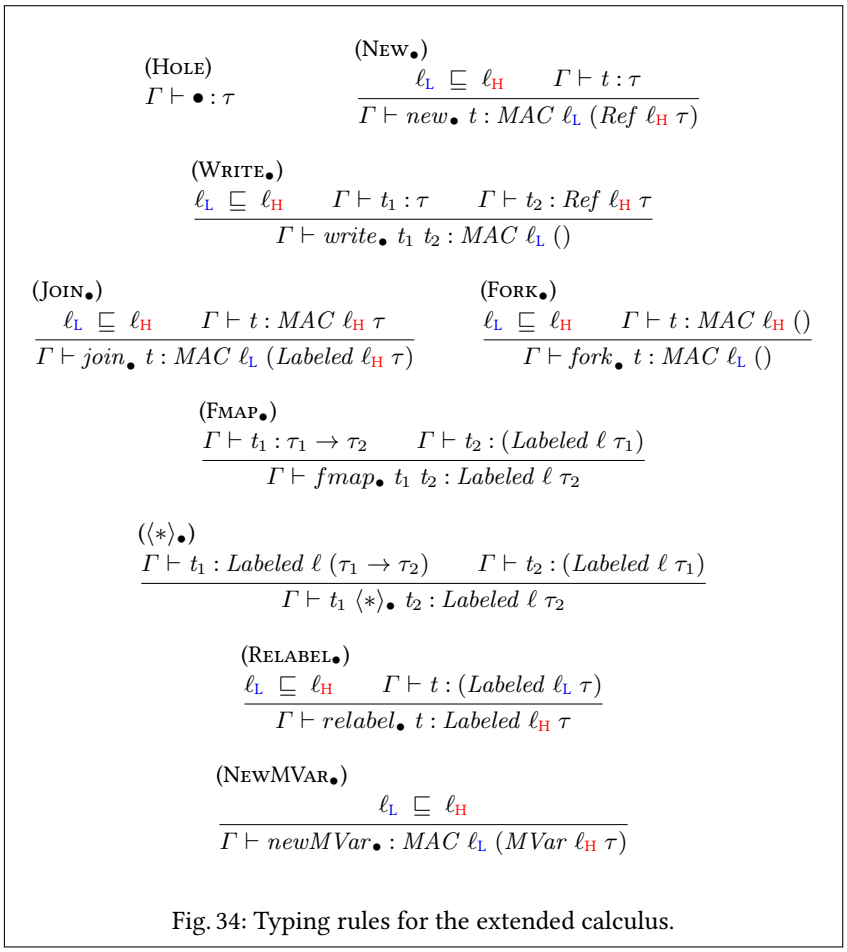


Fig. 34: Typing rules for the extended calculus.

C Typing Rules

Figure 34 gives the typing rules for the extended calculus, i.e., term $\bullet$ and the other $\bullet$-annotated terms introduced to apply erasure in two steps. The typing rule [Hole] assigns an arbitrary type τ to the special term $\bullet$, and the typing rules of the $\bullet$-annotated terms correspond to the rules of the corresponding vanilla terms, which we have given in the form of type-signatures throughout the article. Then, it is straightforward to prove that the erasure function is type-preserving, i.e., if $\Gamma \vdash t : \tau$ then $\Gamma \vdash \varepsilon_{\ell_A}(t) : \tau$. In our machine-checked proof scripts, terms are encoded using a well-typed syntax, which merges the syntax of terms and the typing judgments in a single object.

Paper IV

Based on

Securing Concurrent Lazy Programs Against Information Leakage,

by Marco Vassena, Joachim Breitner and Alejandro Russo,

30th IEEE Computer Security Foundations Symposium.

CHAPTER

THREE

SECURING CONCURRENT LAZY PROGRAMS AGAINST INFORMATION LEAKAGE

Abstract. Many state-of-the-art information-flow control (IFC) tools are implemented as Haskell libraries. A distinctive feature of this language is lazy evaluation. In his influencal paper on why functional programming matters [62], John Hughes proclaims:

> *Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.*

Unfortunately, lazy evaluation makes IFC libraries vulnerable to leaks via the internal timing covert channel. The problem arises due to *sharing*, the distinguishing feature of lazy evaluation, which ensures that results of evaluated terms are stored for subsequent re-utilization. In this sense, the evaluation of a term in a high context represents a *side-effect* that eludes the security mechanisms of the libraries. A naïve approach to prevent that consists in forcing the evaluation of terms before entering a high context. However, this is not always possible in lazy languages, where terms often denote infinite data structures. Instead, we propose a new language primitive, *lazyDup*, which duplicates terms *lazily*. We make the security library **MAC** robust against internal timing leaks via lazy evaluation, by using *lazyDup* to duplicate terms manipulated in high contexts, as they are evaluated. We show that well-typed programs satisfy progress-sensitive non-interference in our lazy calculus with non-strict references. Our security guarantees are supported by mechanized proofs in the Agda proof assistant.

1 Introduction

Information-Flow Control [128] (IFC) scrutinizes source code to track how data of different sensitivity levels (e.g., public or sensitive) flows within a program, and raises alarms when confidentiality might be at stake. There are

several special-purpose compilers and interpreters which apply this technology: *Jif* [99] (based on Java), *flowcaml* [116] (based on Caml and not developed anymore), *Paragon* [28] (based on Java), and *JSFlow* [51] (based on JavaScript). Rather than writing compilers/interpreters, IFC can also be provided as a library in the functional programming language Haskell [75].

Haskell's type system enforces a disciplined separation of side-effect free from side-effectful code, which makes it possible to introduce input and output (I/O) to the language without compromising on its *purity*. Computations performing side-effects are encoded as values of abstract types which have the structure of monads [92]. This distinctive feature of Haskell is exploited by state-of-the-art IFC libraries (e.g., **LIO** [145] and **MAC** [123]) to identify and restrict "leaky" side-effects without requiring changes to the language or runtime.

Another distinctive feature of Haskell is its *lazy evaluation* strategy. This evaluation is non-strict, as function arguments are not evaluated until required by the function, and it performs *sharing*, as the values of such arguments are stored for subsequent uses. In contrast, eager evaluation, also known as *strict evaluation*, reduces function arguments to their denoted values before executing the function.

From a security point of view, it is unclear which evaluation strategy—lazy or strict—is more suitable to preserve secrets. To start addressing this subtlety, we need to consider the interaction between evaluation strategies and covert channels.

Sabelfeld and Sands [130] suggest that lazy evaluation might be intrinsically safer than eager evaluation for leaks produced by termination—as lazy evaluation could skip the execution of *unneeded* non-terminating computations that might involve secrets. In multi-threaded systems, where termination leaks are harmful [141], a lazy evaluation strategy seems to be the appropriate choice.

Unfortunately, although lazy evaluation could "save the day" when it comes to termination leaks, it is also vulnerable to leaks via another covert channel due to sharing. Buiras and Russo [31] described an attack against the **LIO** library [141] where lazy evaluation is exploited to leak information via the *internal timing covert channel* [138]. This covert channel manifests by the mere presence of concurrency and shared resources. It gets exploited by setting up threads to race for a public shared resource in such a way that the secret value affects their timing and hence the winner of the race. **LIO** removes such leaks for public shared-resources which can be identified by the library (e.g., references). Due to lazy evaluation, variables introduced by **let**-bindings and function applications—which are beyond **LIO**'s control[1]—become shared resources and their evaluation affects the threads' timing behavior. Note that the *internal* timing channel leverages the *order* with which threads access the

[1] As a shallow EDSL, **LIO** reuses much of the host language features to provide security (e.g., type-system and variable bindings). This design choice makes the code base small at the price of not fully controlling the features provided by the host language.

```
let ℓ = [1 . . 10000000]
    r = sum ℓ
in do fork^LIO    -- Secret thread
        (do s ← unlabel secret
            when (s ≡ 1 ∧ r ⩾ 10) return ())
      no_ops; no_ops
        -- Public threads
      fork^LIO (do sendPublicMsg (r − r))
      fork^LIO (do no_ops; sendPublicMsg 1)
```

Fig. 1: Lazy evaluation attack.

shared resource, not their execution time, which constitutes a different covert channel, known as the *external* timing covert channel [24, 41]. The attacker model for the external timing covert channel assumes that the attacker has access to an arbitrarily precise stopwatch to measure the wall-clock execution time of instructions and thereby deduce information about secrets. This paper does not address the external timing covert channel, which is a harder problem and for which mitigation techniques exist [5, 168, 169].

Figure 1 shows the lazy evaluation attack. In **LIO**, every thread has a current label which serves a role similar to the *program counter* in traditional IFC systems [157]. The first thread inspects a secret value ($s \leftarrow unlabel\ secret$), which sets the current label to secret. We refer to threads with such current label as *secret threads*. The other spawned threads have their current label set to public, therefore we call them *public threads*. Observe that the variable r hosts an expression that is somewhat expensive to calculate, as it first builds a list with ten million numbers ($\ell = [1 \,..\, 10000000]$) before summing up its elements ($r = sum\ \ell$). Importantly, the variable r is referenced by both the secret and the public threads. Observe that every thread is secure in isolation—the secret thread always returns $()$ and the public threads read no secret. Assume that the expression no_ops is some irrelevant computation that takes slightly longer than half the time it takes to sum up the ten million numbers. Then the public threads race to send a message on a shared-public channel using the function $sendPublicMsg$:

▷ If $s \equiv 1$, then the secret thread has by now evaluated the expression referenced by r, in order to check if $r \geqslant 10$ holds. Due to sharing, the first public thread will not have to re-calculate r and can output 0 almost imediately, while the other public thread is still occupied with no_ops.

▷ If $s \equiv 0$, then the secret thread did not touch r. While the first public thread now has to evaluate r, the second public thread has enough time to perform no_ops and output 1 first.

As a result, the last message on the public channel reveals the secret s. This attack can be magnified to a point where whole secrets are leaked systemati-

cally and efficiently [141]. Similar to **LIO**, other state-of-the-art concurrent IFC Haskell libraries [30, 123] suffer from this attack.

A naïve fix is to force variable r to be fully evaluated before any public threads begin their execution. This works, but it defeats the main purpose of lazy evaluation, namely to avoid evaluating unneeded expressions. Furthermore, it is not always possible to evaluate expressions to their denoted value. Haskell programmers like to work with infinite structures, even though only finite approximation of them are actually used by programs. For example, if variable ℓ in Figure 1 were the list $[1 \,/\, n \mid n \leftarrow [1\,..]]$ of reciprocals of all natural numbers and r the sum of those bigger than one millionth ($r = sum\ (takeWhile\ (\geqslant 1\mathrm{e}{-6})\ \ell)$). The evaluation of r uses only a finite portion of ℓ, so the modified program still terminates. But naïvely forcing ℓ to normal form would hang the program. This demonstrates that simply forcing evaluation as a security measure is unsatisfying, as it can introduce divergence and thus change the meaning of a program.

Instead, we present a novel approach to explicitly control sharing at the language level. We design a new primitive called $lazyDup$ which *lazily* duplicates unevaluated expressions. The attack in Figure 1 can then be neutralized by replacing r with $lazyDup\ r$ in the secret thread, which will then evaluate its own copy of r, without affecting the public threads.

To the best of our knowledge, this work is the first one to formally address the problem of internal timing leaks via lazy evaluation. In summary, our contributions are:

- We present $lazyDup$, a primitive to restrict sharing in lazy languages with mutuable references.
- By injecting $lazyDup$ when spawning threads, we demonstrate that internal timing leaks via lazy evaluation are closed. The primitive $lazyDup$ is not only capable to secure **MAC** against lazy leaks, but also a wide range of other security Haskell libraries (e.g., **LIO** and **HLIO**).
- We prove that well-typed programs satisfy progress-sensitive noninterference (PSNI) for a wide-range of deterministic schedulers. However, for ease of exposition in this article, we focus only on a round-robin scheduler—the same scheduler used in GHC's runtime system.[2] Our non-interference claims are supported by mechanized proofs in the Agda proof assistant [103] and are parametric on the chosen (deterministic) scheduler.
- As a by-product of interest for the programming language community, we provide—to the best of our knowledge—the first operational semantics for lazy evaluation with mutable references.

This paper is organized as follows. Section 2 provides a brief overview on **MAC**. Section 3 describes our formalization for a concurrent non-strict calculus with sharing that also includes references. Primitive $lazyDup$ is described in

[2] The Glasgow Haskell Compiler (GHC) is a state-of-the-art, industrial-strength, open source Haskell compiler.

```
   -- Abstract types
data Labeled ℓ τ
data MAC ℓ τ
   -- Monadic structure for computations
instance Monad (MAC ℓ)
   -- Core operations
label    :: ℓL ⊑ ℓH ⇒ τ → MAC ℓL (Labeled ℓH τ)
unlabel  :: ℓL ⊑ ℓH ⇒ Labeled ℓL τ → MAC ℓH τ
fork^MAC :: ℓL ⊑ ℓH ⇒ MAC ℓH () → MAC ℓL ()
```

Fig. 2: Core API for **MAC**

Section 4. Section 5 shows how *lazyDup* can remove leaks via lazy evaluation and Section 6 provides the corresponding security guarantees. Related work is given in Section 7 and Section 8 concludes.

2 Overview of MAC

To set the stage of the work at hand, we briefly introduce the relevant aspects of the **MAC** IFC library [123].

Security lattice. The sensitivity of data is indicated by labels. These are partially ordered by $\sqsubseteq$ and form a security lattice [37]. Concretely, $\ell_1 \sqsubseteq \ell_2$ holds if data labeled with label ℓ_1 is allowed to flow to entities labeled with ℓ_2. Although **MAC** is parameterized on the security lattice, for simplicity we focus on the classic two-point lattice where the label H denotes secret (high) data, the label L denotes public (low) data, and $H \not\sqsubseteq L$ is the only disallowed flow. In **MAC**, each label is represented as an abstract data type. To improve readability, subscripts on label metavariables hint at their relationship, e.g., if ℓ_L and ℓ_H appear together, then $\ell_L \sqsubseteq \ell_H$ holds.

Security Types. Figure 2 shows the core of **MAC**'s API. The abstract type $Labeled\ \ell\ \tau$ classifies data of type τ with a security label ℓ. For example $creditCard :: Labeled\ H\ Int$ represents a sensitive integer and $weather :: Labeled\ L\ String$ represents a public string. The abstract type $MAC\ \ell\ \tau$ denotes a (possibly) side-effectful secure computation which handles information at sensitivity level ℓ and yields a value of type τ as a result. Importantly, a $MAC\ \ell\ \tau$ computation enjoys a monadic structure, i.e., it is built by the two fundamental operations $return :: \tau \to MAC\ \ell\ \tau$ and $(\gg\!=) :: MAC\ \ell\ \tau \to (\tau \to MAC\ \ell\ \tau') \to MAC\ \ell\ \tau'$ (called "bind"). The operation $return\ x$ produces a computation that returns the value denoted by x without causing side-effects. The function $(\gg\!=)$ is used to *sequence* computations and their corresponding side-effects. Specifically, $m \gg\!= f$ takes the *result* of running the computation m and passes it to the function f, which then returns a second computation to run. Haskell provides syntactic sugar for monadic computations known as

```
impl :: Labeled H Bool → MAC H (Labeled L Bool)
impl secret = do
  bool ← unlabel secret
  if bool then label True
          else label False
```

Fig. 3: Implicit flows are ill-typed ($H \not\sqsubseteq L$).

do-notation. For instance, the program $m \gg\!= \lambda x \rightarrow return\ (x + 1)$, which adds 1 to the value produced by m, can be written as follows.

$$\begin{aligned}&\textbf{do}\ x \leftarrow m\\ &\quad return\ (x + 1)\end{aligned}$$

Flows of information. Abstractly, the side-effects of a $MAC\ \ell\ \tau$ computation involve either reading or writing data. We need to ensure that these actions respect the flows of information that are permitted by the security lattice. The functions *label* and *unlabel* allow $MAC\ \ell\ \tau$ computations to securely interact with labeled expressions, which are the simplest kind of resources available in **MAC**. If a $MAC\ \ell_{\mathrm{L}}$ computation writes data into a sink, the computation needs to have at most the sensitivity of the sink itself. This restriction, known as *no write-down* [19], preserves the sensitivity of data handled by the $MAC\ \ell_{\mathrm{L}}$-computation. The function *label* creates a fresh, labeled value. From the security point of view, this action corresponds to allocating a fresh location in memory and immediately writing a value into it—hence the no write-down principle applies. The type signature of *label* has a *type constraint* before the symbol $\Rightarrow$, which is a property that types must follow. The constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that, when calling $label\ x$, the level of the computation ℓ_{L} is no more confidential than the sensitivity ℓ_{H} of the labeled value that it creates. In contrast, a computation $MAC\ \ell_{\mathrm{H}}\ \tau$ is only allowed to read labeled values at most as sensitive as ℓ_{H}. This restriction is known as *no read-up* [19] and gets enforced by the constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ in the type signature of *unlabel*. This paper focuses on labeled expression, but **MAC** provides additional side-effecting primitives for exception handling, network communication, references, and synchronization primitives [123].

Implicit flows. The interaction between the type of a $MAC\ \ell$-computation and the no write-down restriction makes an implicit flow ill-typed. Figure 3 shows a program that attempts to *implicitly* leak a Boolean secret, which is correctly rejected by the compiler. In order to branch on sensitive data, a program needs first to unlabel it, which forces the computation to be of type $MAC\ H\ \tau$, for some type τ. Regardless of which branch is taken, the computation is at level H and cannot therefore write into public data due to the *no write-down* restriction. Trying to do so, as shown in Figure 3, incurs in a violation of the security policy and a type error! Observe that the application of *label* is rejected since its type constraint cannot be satisfied, i.e., $H \not\sqsubseteq L$.

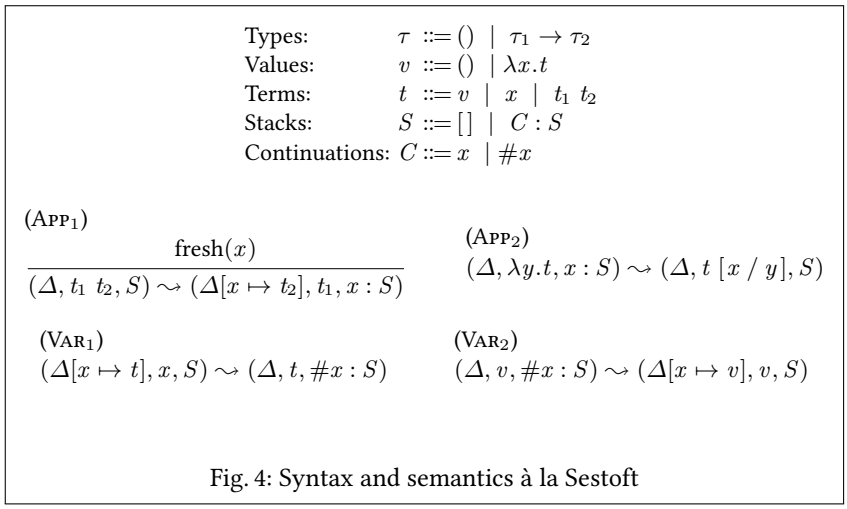


Fig. 4: Syntax and semantics à la Sestoft

Concurrency. The mere possibility to run (conceptually) simultaneous $MAC\ \ell$ computations provides attackers with new tools to bypass security checks. In particular, threads introduce the *internal timing covert channel* described in the introduction. Furthermore, it considerably magnifies the bandwidth of the termination covert channel, where secrets are learned by observing the terminating behavior of threads [141]. To securely support concurrency, **MAC** forces programmers to decouple computations which depend on sensitive data from those performing public side-effects. In this manner, non-terminating loops based on secrets cannot affect the outcome of public events. In this light, the type signature of $fork^{\text{MAC}}$ in Figure 2 only allows spawning threads, i.e., a secure computation with type $MAC\ \ell_{\text{H}}\ ()$, which are at least as sensitive as the current computation, i.e., $MAC\ \ell_{\text{L}}\ ()$. It is secure to do so because that decision depends on less sensitive data ($\ell_{\text{L}} \sqsubseteq \ell_{\text{H}}$).

3 Lazy Calculus

In order to rigorously analyze the information leaks introduced by sharing, we need to build on top of a formal semantics that is operationally precise enough to make sharing observable. The default choice for such a semantics is Launchbury's "Natural Semantics for lazy evaluation" [73], where the structure of the heap is explicit and sharing, as well as cyclic data structures, are manifestly visible. The heap is a partial map from names to terms. This representation is still more abstract than other formalisations such as the Spineless Tagless G-machine (STG) [109], which concerns itself with pointers and memory representation, and is the basis of the Haskell implementation GHC [88]. That much operational detail would only clutter this work, and in terms of lazy evaluation, Launchbury's semantics is a suitable model of the actual implementation.

This work will have to address concurrency, for which a big-step semantics such as Launchbury's is unsuitable for. Therefore, we build on Sestoft's

rendering of Launchbury's semantics as an abstract machine with small-step semantics [135]. Here, a judgement $(\Delta, t, S) \rightsquigarrow (\Delta', t', S')$ indicates that a configuration consisting of a current expression t, a heap Δ, and a stack S takes one step to the configuration on the right hand side of the arrow.

The rules in Figure 4 describe the transitions of the abstract machine for the standard syntactical constructs. Rule ($\textsc{App}_1$) initiates a function call. Since we work in a lazy setting, the function argument t_2 is not evaluated at this point. Instead, it is stored on the heap as a *thunk*, i.e., an unevaluated expression, under a fresh name x with regard to the whole configuration—which corresponds to allocating memory. The machine proceeds to evaluate the function expression t_1 to a lambda expression. Then, rule ($\textsc{App}_2$) takes over and substitutes the name of the argument x, which is found on the stack, into the body t of the lambda expression. The argument x may, however, need to be evaluated at some point. Rule ($\textsc{Var}_1$) finds the corresponding thunk t on the heap and, after leaving an *update marker* $\#x$ on the stack, begins to evalute the thunk—intuitively, this marker indicates that when the evaluation of the current term finishes, the denoted value gets stored in x. During evaluation, x is removed from the heap. If the evaluation of t required the value of x, the machine would get stuck. This effect is desired: if the binding for x were to remain on the heap, evaluation would simply start to run in circles. Removing the variable from the heap, a technique called *blackholing*, makes this error condition detectable. When the machine reduces the thunk to a value v, rule ($\textsc{Var}_2$) pops the update marker from the stack and puts x back on the heap, but now referencing to the value v. Every future use of x will use v directly instead of re-calculating it. This *updating* operation is the crucial step to implement sharing behavior.

We simplified Sestoft's presentation of the semantics in a few ways to remove aspects not relevant for the discussion at hand and to facilitate our machine-checked proofs in Agda: *i)* our syntax does not include mutual recursive **let** expressions; *ii)* in contrast to Sestoft and Launchbury, we allow non-trivial arguments in function application, i.e., our terms are not necessarily in Administrative Normal Form (ANF). In that manner, a non-recursive let expression such as $\textbf{let}\ x = t_1\ \textbf{in}\ t_2$ can be expressed as $(\lambda x.t_2)\ t_1$; *iii)* although omitted in this presentation, our formalism sports types with multiple values (e.g., Boolean expressions) and the corresponding case-analysis clause (e.g., **if-then-else**) by using the rules found in [27].

3.1 Security Primitives

We now extend this standard calculus with the security primitives of **MAC** as shown in Figure 5. The new type $\mathit{Labeled}\ \ell\ \tau$ consists of pure values $t :: \tau$ wrapped in $\mathit{Labeled}$, and annotates them with the security level ℓ. We call $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ \ell\ \mathit{Int}$ a pure, side-effect free labeled-ℓ resource with content 42. We introduce a further form of labeled resource, namely references, in the next section. The semantics rules in Figure 5 are fairly straight-forward and follow the pattern seen in Figure 4. It is worth noting that thanks to the static nature of **MAC**, no run-time checks are needed to prevent insecure flows

$$
\begin{array}{lll}
\text{Labels:} & \ell & \\
\text{Types:} & \tau & ::= \cdots \mid \mathit{MAC}\ \ell\ \tau \mid \mathit{Labeled}\ \ell\ \tau \\
\text{Values:} & v & ::= \cdots \mid \mathit{return}\ t \mid \mathit{Labeled}\ t \\
\text{Terms:} & t & ::= \cdots \mid t_1 \gg\!= t_2 \mid \mathit{label}\ t \mid \mathit{unlabel}\ t \\
\text{Continuations:} & C & ::= \cdots \mid \gg\!= t \mid \mathit{unlabel}
\end{array}
$$

$$
\frac{}{(\Delta, \mathit{label}\ t, S) \rightsquigarrow (\Delta, \mathit{return}\ (\mathit{Labeled}\ t), S)}\ (\textsc{label})
$$

$$
\frac{}{(\Delta, \mathit{unlabel}\ t, S) \rightsquigarrow (\Delta, t, \mathit{unlabel} : S)}\ (\textsc{unlabel}_1)
$$

$$
\frac{}{(\Delta, \mathit{Labeled}\ t, \mathit{unlabel} : S) \rightsquigarrow (\Delta, \mathit{return}\ t, S)}\ (\textsc{unlabel}_2)
$$

$$
\frac{}{(\Delta, t_1 \gg\!= t_2, S) \rightsquigarrow (\Delta, t_1, \gg\!= t_2 : S)}\ (\textsc{bind}_1)
\qquad
\frac{}{(\Delta, \mathit{return}\ t, \gg\!= t_2 : S) \rightsquigarrow (\Delta, t_2\ t, S)}\ (\textsc{bind}_2)
$$

Fig. 5: Security primitives

of information in these rules. We remark that the constructor *Labeled* is not available to the user, who can only use *label* (*unlabel*) to create (inspect) labeled resources. Besides these primitives, the user can create computations using the standard monad operations *return* and $\gg\!=$.

The actual **MAC** implementation knows even more labeled resources (e.g., network) [123]. **MAC** requires no modification to Haskell's type system in order to handle labels: each label is defined as an *empty type*, i.e., a type that has no value, and labeled resources (type *Labeled*) use labels as *phantom types*, i.e., a type parameter that only carries the sensitivity of data at the type-level.

3.2 References

We now extend the abstract machine with mutable references, a feature available in **MAC** to boost the performance of secure programs [123]. References live in the *memory* M, which is simply a list of variables, added as a component of the program configuration—see Figure 6. The address of a memory cell is its index in this list. The memory $M[n \mapsto x]$ is M with its n-th cell changed to refer to x. Observe that the memory M and the heap Δ are two distinct syntactic categories and that, while the latter contains arbitrary terms and enjoys sharing, the former merely contains pointers to the heap. A *labeled* reference is represented as a value $\mathit{Ref}\ n :: \mathit{Ref}\ \ell\ \tau$ where n is the address of the n-th memory cell, which contains a variable (a "pointer") to some term $t :: \tau$ on the heap[3]. Only secure computations can manipulate these *labeled references* using the following secure primitives. Observe that the types are restricted

[3] **MAC**'s implementation of labeled reference is a simple wrapper around Haskell's type *IORef*. However, we denote references as a simple index into the labeled memory. This design choice does not affect our results.

$$
\begin{array}{lll}
\text{Configuration:} & c & ::= \langle M, \Delta, t, S\rangle \\
\text{Memory:} & M & ::= [\,] \mid x : M \\
\text{Addresses:} & n & \in \mathbb{N} \\
\text{Types:} & \tau & ::= \cdots \mid \mathit{Ref}\ \ell\ \tau \\
\text{Values:} & v & ::= \cdots \mid \mathit{Ref}\ n \\
\text{Terms:} & t & ::= \cdots \mid \mathit{new}\ t \mid \mathit{read}\ t \mid \mathit{write}\ t_1\ t_2 \\
\text{Continuations:} & C & ::= \cdots \mid \mathit{read} \mid \mathit{write}\ t
\end{array}
$$

(Lift)
$$\frac{(\Delta, t, S) \rightsquigarrow (\Delta', t', S')}{\langle M, \Delta, t, S\rangle \longrightarrow \langle M, \Delta', t', S'\rangle}$$

(New)
$$\frac{n = |M| \qquad \mathrm{fresh}(x) \qquad M' = M[n \mapsto x] \qquad \Delta' = \Delta[x \mapsto t]}{\langle M, \Delta, \mathit{new}\ t, S\rangle \longrightarrow \langle M', \Delta', \mathit{return}\ (\mathit{Ref}\ n), S\rangle}$$

(Write$_1$)
$$\langle M, \Delta, \mathit{write}\ t_1\ t_2, S\rangle \longrightarrow \langle M, \Delta, t_1, \mathit{write}\ t_2 : S\rangle$$

(Write$_2$)
$$\frac{\mathrm{fresh}(x) \qquad M' = M[n \mapsto x] \qquad \Delta' = \Delta[x \mapsto t]}{\langle M, \Delta, \mathit{Ref}\ n, \mathit{write}\ t : S\rangle \longrightarrow \langle M', \Delta', \mathit{return}\ (), S\rangle}$$

(Read$_1$)
$$\langle M, \Delta, \mathit{read}\ t, S\rangle \longrightarrow \langle M, \Delta, t, \mathit{read} : S\rangle$$

(Read$_2$)
$$\langle M, \Delta, \mathit{Ref}\ n, \mathit{read} : S\rangle \longrightarrow \langle M, \Delta, \mathit{return}\ M[n], S\rangle$$

Fig. 6: Syntax and semantics for references.

according to the *no read-up* and *no write-down* restrictions—like those of label and $\mathit{unlabel}$.

The extended semantics is represented as the relation $c \longrightarrow c'$ which extends $\rightsquigarrow$ via [Lift]—see Figure 6. Rule [New] allocates the second argument on the heap with a fresh name x, extends the memory with a new pointer to x and returns a reference to it. Rule [Write$_1$] evaluates its first argument to a reference and rule [Write$_2$] overrides the memory cell with a pointer to a newly allocated heap entry, just like new. The two [Read]-rules retrieve a pointer from memory. To the best of our knowledge, this is the first published operational semantics that models both lazy evaluation and mutable references, and although we constructed it using standard techniques, we would like to point out a crucial subtlety.

A naïve model might omit the extra memory M, let a reference simply contain a variable on the heap ($t ::= \cdots \mid \mathit{Ref}\ x$), and use the transition rule

$$
\begin{aligned}
&\mathit{new} &&:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \to \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Ref}\ \ell_{\mathrm{H}}\ \tau)\\
&\mathit{read} &&:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Ref}\ \ell_{\mathrm{L}}\ \tau \to \mathit{MAC}\ \ell_{\mathrm{H}}\ \tau\\
&\mathit{write} &&:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \to \mathit{Ref}\ \ell_{\mathrm{H}}\ \tau \to \mathit{MAC}\ \ell_{\mathrm{L}}\ ()
\end{aligned}
$$

Fig. 7: API of memory operations.

$\langle \Delta, \mathit{Ref}\ x, \mathit{write}\ t : S\rangle \longrightarrow \langle \Delta[x \mapsto t], \mathit{return}\ (), S\rangle$. This transition interacts badly with sharing, as shown by the following program.

$$
\begin{aligned}
&\textbf{do}\ r \leftarrow \mathit{new}\ (1+1)\\
&\quad x \leftarrow \mathit{read}\ r\\
&\quad \mathit{write}\ r\ 1\\
&\quad \textbf{if}\ (x \equiv 2)\ \textbf{then}\ \mathit{return}\ \texttt{"☺"}\\
&\qquad\qquad\ \ \textbf{else}\ \ \mathit{return}\ \texttt{"☹"}
\end{aligned}
$$

Clearly, we expect the program to return "☺", but it returns "☹" with the naïve semantics! The new statement allocates a new variable x, binds it to $1+1$, and returns the reference $\mathit{Ref}\ x$. The next read statement brings variable x into scope, which is pure and we expect its denoted value to stay the same. However, under the naïve semantics, the following write statement changes x to 1 and therefore chaos ensues.

The solution is to add an extra layer of indirection, and distinguish between the mutable memory cells that make up a reference, and the heap locations that—although changed in [VAR$_2$]—are conceptually constant. We chose to keep track of them separately in the memory M and the heap Δ since we found that it makes formal reasoning easier. It is also viable to keep both on the heap, and just be disciplined as to which variables denote references and which denote values and thunks—this design choice would be closer to GHC's runtime, where both pure data and mutable references are addressed by pointers into a single heap.

3.3 Concurrency

Finally, we extend our language with concurrency in the form of threads whose execution interleave.[4] We consider *global configurations* of the form $\langle M, \Delta, T_s\rangle$, where thread pool T_s consists of a list of threads—see Figure 8. A thread (t, S) is an *interrupted* secure computation, consisting of a control term t and a stack S. Within a global configuration, threads are identified by their position in the thread pool. For simplicity and brevity, the concurrent calculus features a Round Robin scheduler, the same kind of scheduler used by GHC's run-time system[5]—however, our results and semantics generalize to a wide range of deterministic schedulers. In the following, we omit the scheduler from the configuration and from the semantics rules for space reasons.

[4] **MAC** provides also synchronization variables [123], which we omit here.

[5] `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Scheduler`

$$
\begin{array}{lll}
\text{Thread pool:} & T_s & ::= [\,] \mid (t, S) : T_s \\
\text{Configuration:} & c & ::= \langle M, \Delta, T_s \rangle \\
\text{Terms:} & t & ::= \cdots \mid \mathit{fork}\ t
\end{array}
$$

(SEQ)

$$\frac{\langle M_1, \Delta_1, t_1, S_1 \rangle \longrightarrow \langle M_2, \Delta_2, t_2, S_2 \rangle \qquad T_s' = T_s[n \mapsto (t_2, S_2)]}{\langle M_1, \Delta_1, T_s[n \mapsto (t_1, S_1)] \rangle \hookrightarrow \langle M_2, \Delta_2, T_s' \rangle}$$

(FORK)

$$\frac{m = |T_s| \qquad T_s' = T_s[n \mapsto (\mathit{return}\ (), S)]}{\langle M, \Delta, T_s[n \mapsto (\mathit{fork}\ t, S)] \rangle \hookrightarrow \langle M, \Delta, T_s'[m \mapsto (t, [\,])] \rangle}$$

Fig. 8: Syntax and Semantics of the concurrent calculus.

Figure 8 describes the two rules under which a global configuration c_1 steps to c_2 (written $c_1 \hookrightarrow c_2$). In both rules ([SEQ] and [FORK]), thread n is executed—the scheduler actually *deterministically* chooses which thread to run, which is retrieved from the thread pool T_s. In rule [SEQ], the selected thread, i.e., (t_1, S_1), takes a sequential step that is paired with the current memory and heap: ($\langle M_1, \Delta_1, t_1, S_1 \rangle \longrightarrow \langle M_2, \Delta_2, t_2, S_2 \rangle$). The global configuration is then updated accordingly to the final sequential configuration; in particular, the thread pool is updated with the reduced thread, i.e., $T_s[n \mapsto (t_2, S_2)]$. In rule [FORK], the selected thread spawns a thread—note that term $\mathit{fork}\ t$ is stuck in the sequential semantics and rule [SEQ] does not apply. The new thread is assigned the fresh identifier $m = |T_s|$—thread pool T_s contains threads $0 \ldots |T_s| - 1$. Lastly, the thread pool is updated with the parent thread, appropriately reduced to $(\mathit{return}\ (), S)$, and by inserting the new thread initialized with an empty stack, i.e., $(t, [\,])$, at position m.

Note that a thread that tries to evaluate a variable x that is already under evaluation by another thread will not find this variable on the heap, due to the blackholing mechanism explained earlier. The thread is now blocked, guaranteeing that, even in the concurrent setting, every thunk will only be evaluated at most once. This mechanism is consistent with the operational semantics used by Finch et al. [12].

4 Duplicating Thunks

This section presents one of our main contributions: a primitive, called *lazyDup*, to prevent sharing. Given a term t, evaluating $\mathit{lazyDup}\ t$ will *lazily* create a copy of t. The laziness is necessary in order to duplicate cyclic or infinite data structures without sending the program into a loop. We first present the basic semantics of *lazyDup* and then describe how we handle references.

Terms: $t ::= \cdots \mid \mathit{lazyDup}\ t$

$$\frac{\neg\ \mathit{isVar}\ (t) \qquad \mathrm{fresh}(x)}{\langle \Delta, \mathit{lazyDup}\ t, S\rangle \rightsquigarrow \langle \Delta[x \mapsto t], \mathit{lazyDup}\ x, S\rangle}\ (\textsc{LazyDup}_1)$$

$$\frac{x \mapsto t\ \in\ \Delta \qquad \mathrm{fresh}(x')}{\langle \Delta, \mathit{lazyDup}\ x, S\rangle \rightsquigarrow \langle \Delta[x' \mapsto [\![t]\!]^{\varnothing}], x', S\rangle}\ (\textsc{LazyDup}_2)$$

$$\begin{aligned}
[\![t_1\ t_2]\!]^B &= [\![t_1]\!]^B\ [\![t_2]\!]^B \\
[\![\lambda x.t]\!]^B &= \lambda x.[\![t]\!]^{B \cup \{x\}} \\
[\![()]\!]^B &= () \\
[\![x]\!]^B &= \begin{cases} x & \text{if } x\ \in\ B \\ \mathit{lazyDup}\ x & \text{if } x\ \notin\ B \end{cases} \\
[\![\mathit{lazyDup}\ t]\!]^B &= \mathit{lazyDup}\ t
\end{aligned}$$

Fig. 9: Syntax and semantics of $\mathit{lazyDup}$.

4.1 Semantics

Figure 9 extends the syntax and the semantics of the calculus with $\mathit{lazyDup}$. The rule [$\textsc{LazyDup}_1$] ensures that the argument of $\mathit{lazyDup}$ is a variable, if that is not already the case. The interesting rule is [$\textsc{LazyDup}_2$], which evaluates $\mathit{lazyDup}\ x$ and copies the expression t referenced by x. This closes the covert channel represented by x, but it is insufficient, as t might mention further variables. Therefore, $\mathit{lazyDup}$ has to descent into t, and handle these as well. But instead of immediately duplicating the terms referenced by those variables, we simply wrap them in a call to $\mathit{lazyDup}$—this is the eponymous laziness.

Figure 9 shows (some of) the equations of the function $[\![t]\!]^B$ which implements this. It homomorphically traverses the tree t, while keeping track of the set of bound variables in its parameter B. Ground values and bound variables are left alone. When $\mathit{lazyDup}$ finds a free variable, i.e., one not in B, it wraps it with a call to $\mathit{lazyDup}$ as intended. In the following, we omit the superscript B when irrelevant. Finally, if $[\![\cdot]\!]$ comes across a call to $\mathit{lazyDup}$, it does not traverse further, as the existing $\mathit{lazyDup}$ already takes care of the duplication. In fact, without this case, evaluating expression $\mathit{lazyDup}\ (\mathit{lazyDup}\ t)$ would send the program into an infinite loop. We conjecture that introducting $\mathit{lazyDup}$ does not change the termination behavior of programs. Note that the term $[\![t]\!]$ has at most one call to $\mathit{lazyDup}$ wrapped around each free variable.

Values: $v ::= \cdots \mid \mathit{DRef}\ m$

$$[\![\mathit{Ref}\ n]\!]^B = \mathit{DRef}\ n$$
$$[\![\mathit{DRef}\ n]\!]^B = \mathit{DRef}\ n$$

(READ^D)
$$\langle M, \Delta, \mathit{DRef}\ n, \mathit{read} : S\rangle \longrightarrow \langle M, \Delta, \mathit{return}\ (\mathit{lazyDup}\ M[n], S)\rangle$$

(WRITE^D)
$$\frac{\text{fresh}(x) \qquad M' = M[n \mapsto x] \qquad \Delta' = \Delta[x \mapsto t]}{\langle M, \Delta, \mathit{DRef}\ n, \mathit{write}\ t : S\rangle \longrightarrow \langle M', \Delta', \mathit{return}\ (), S\rangle}$$

Fig. 10: Duplicate-on-read memory operations.

4.2 References

Duplicating references requires particular care. To illustrate this, consider what does not work. We cannot leave references alone ($[\![\mathit{Ref}\ n]\!] = \mathit{Ref}\ n$) because thunks can be passed through the reference and open a new leaking channel. We cannot either duplicate the reference and the term it currently references since this would change the semantics of mutable references. More concretely, consider a $\mathit{Ref}\ n$ with $M[n] = x$ and $\Delta(x) = t$. Assume we duplicate t to $\Delta(y) = [\![t]\!]$ for a fresh name y and let $[\![\mathit{Ref}\ n]\!] = \mathit{Ref}\ n'$ for a fresh memory cell n', such that $M[n' \mapsto y]$. If later $\mathit{Ref}\ n$ gets updated with the value 42, i.e., $M[n \mapsto z]$ with $\Delta(z) = 42$, then this change would be invisible to $\mathit{Ref}\ n'$, which would still refer to $[\![t]\!]$ through variable y. This is bad, as $\mathit{lazyDup}$ is not supposed to change the observable semantics of the program!

Crucially, we need to propagate any write operation on the original reference to the duplicated reference. One manner to achieve that is to have both references pointing to the same memory location but carefully preventing leaks from reading this shared resource. In this light, we introduce a new variant of reference, called *duplicate-on-read reference*, which is represented by $\mathit{DRef}\ n$.[6] When reading from a $\mathit{DRef}\ n$, we wrap the read value in a call to $\mathit{lazyDup}$, as shown in Figure 10, while write operations on a duplicate-on-read reference are executed as usual. Function $[\![\cdot]\!]$ does not need to follow references and duplicate their content, but simply turns them into duplicate-on-read-references. In this sense, we apply $\mathit{lazyDup}$ lazily to reference: the duplication is suspended and continues when the reference is read.

5 Securing MAC

We now pinpoint the vulnerability leveraged by the attack sketched in the introduction and show how to modify **MAC** to close it using $\mathit{lazyDup}$. It turns out that one careful modification to the [FORK] rule suffices. This change, highlighted in greeen in Figure 11, ensures that when we create a new thread to

[6] The same approach applies to synchronization variables.

(FORK)

$$\frac{|T_s| = m \qquad T_s' = T_s[n \mapsto (return\ (), S)]}{\langle M, \Delta, T_s[n \mapsto (fork\ t, S)]\rangle \hookrightarrow \langle M, \Delta, T_s'[m \mapsto (\ lazyDup\ \ t, [\,])]\rangle}$$

Fig. 11: The green patch secures **MAC**.

evaluate t, it will work on a lazily duplicated copy of t, i.e., $lazyDup\ t$. As a result, each thunk shared between the parent and the child thread gets lazily duplicated: the parent thread works on the original thunk, while the child thread works on a copy.[7]

Let us trace how our proposal fixes the leak shown in Figure 1. Let t be the code of the secret thread. When it is spawned, $lazyDup\ t$ is added to the thread pool. Note that the critical resource that causes the leak, namely variable r, is a free variable of t. As the secret thread executes $lazyDup\ t$, the occurrence of r in the code is replaced by $lazyDup\ r$ (rule [LAZYDUP$_2$]). Therefore, if $s \equiv 1$, the thread duplicates r before evaluating it, leaving r itself alone, just like when $s \equiv 0$ and the secret thread does not touch r at all. As a result, the timing behavior of public threads, i.e., the order with which they output a message on the public channel, is unaffected by the value of the secret s and the internal timing leak is closed.

Observe that $lazyDup$ *conservatively* avoids sharing between secret and public threads. In principle, it is acceptable for a secret thread to evaluate and update a thunk if that action does not depend on the secret—for example if that happens before any sensitive command such as $unlabel$. Assessing whether this is the case requires sophisticated program analysis techniques, which are beyond the scope of this paper. On the other hand, sharing from public to secret threads is always secure, and in fact $lazyDup$ allows for this "write-up" behavior: if, due to lucky scheduling, the public thread finishes evaluating r before the secret thread looks at it, then the latter will see the fully evaluated term and securily enjoy the benefits of sharing.

The primitive $lazyDup$ is capable of securing **LIO** as well even though it has to be used differently—see Appendix A for more details.

6 Security Guarantees

In this section, we show that our calculus satisfies *progress sensitive noninterference* (PSNI). We start by describing our proof technique, based on *term erasure*. To facilitate reasoning, we proceed to decorate our calculus with labels that keep track of the security level of terms stored in memory, heaps and configurations. We then prove PSNI for the decorated calculus and conclude that **MAC**

[7] It is secure to avoid duplication whenever the parent and the child thread share the same security level, which are both statically known in **MAC**, see Figure 2. Since the label of the child thread (ℓ_H) is at least as sensitive as that of the parent, i.e., $\ell_L \sqsubseteq \ell_H$, we only have to use $lazyDup$ if $\ell_L \sqsubset \ell_H$.

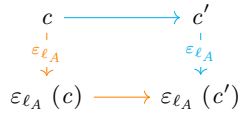


Fig. 12: Single-step simulation.

is likewise secure by establishing a mutual simulation relation with the vanilla (undecorated) calculus.

6.1 Term Erasure

Term erasure is a technique to prove noninterference in functional programs [76] and IFC libraries (e.g., [30,55,141,145,153]). It relies on an erasure function, which we denote by ε_{ℓ_A}. This function rewrites data above the attacker's security level, denoted by label ℓ_A, to the special syntactic construct $\bullet$. At the core, this technique establishes a simulation between reductions of configurations and reductions of their erased counterparts. Figure 12 shows that erasing sensitive data from a configuration c and then taking a step (orange path) yields the same configuration as first taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. If the configuration c were to leak sensitive data into a non-sensitive resource, then it will remain in $\varepsilon_{\ell_A}(c')$. The same data would be erased in $\varepsilon_{\ell_A}(c)$ and the diagram would not commute.

6.2 Decorated Calculus

The erasure proof technique was conceived to work on dynamic IFC approaches [76], where security labels are attached to terms. Applying term erasure to **MAC**, where labels are parts of the types instead of the terms, demands to extend our calculus with extra information about the sensitivity nature of terms. As in similar work [152, 153], we annotate terms with their type and make the erasure function *type-driven*. The annotated term $t :: \tau$ denotes that the term t has type τ. We likewise decorate configurations, heaps, memories, stacks, and continuations with labels.

Figure 13 summarizes the main changes required to decorate our calculus. A pure configuration $\langle \Delta^\ell, t, S^\ell \rangle$, labeled with ℓ, consists of a labeled heap Δ^ℓ, and a labeled stack S^ℓ. An ℓ-labeled heap Δ^ℓ can be accessed by ℓ-labeled variables, e.g., x^ℓ. An ℓ-labeled stack contains exclusively ℓ-labeled continuations, which involve only ℓ-labeled variables, i.e., continuations x^ℓ and $\#x^\ell$. Furthermore an ℓ-labeled heap contains terms that can be evaluated only by threads at level ℓ. A sequential configuration $\langle \Sigma, \Gamma, t, S^\ell \rangle$ labeled with ℓ, consists of a store Σ, a current term t, an heap map Γ, and a labeled stack S^ℓ. An ℓ-labeled configuration denotes a computation of type $MAC\ \ell\ \tau$, for some type τ. Note that this does not necessarily mean that term t is a MAC computation—when the configuration steps the current term is changed with the next redex, which might have a completely different type. Instead, the combination of current

$$
\begin{array}{lrcl}
\text{Pure conf. } \ell\text{:} & c & ::= & (\Delta^{\ell}, t, S^{\ell}) \\
\text{Seq. conf. } \ell\text{:} & c & ::= & \langle \Sigma, \Gamma, t, S^{\ell} \rangle \\
\text{Heap map:} & \Gamma & \in & (\ell : \mathit{Label}) \to \mathit{Heap}\ \ell \\
\text{Store:} & \Sigma & \in & (\ell : \mathit{Label}) \to \mathit{Memory}\ \ell \\
\text{Memory } \ell_{\mathrm{H}}\text{:} & M & ::= & \cdots \mid x^{\ell_{\mathrm{L}}} : M \\
\text{Terms } \tau\text{:} & t & ::= & \cdots \mid x^{\ell} \\
\text{Cont. } \ell\text{:} & C & ::= & \cdots \mid x^{\ell} \mid \#x^{\ell} \\
\text{Conc. conf.:} & c & ::= & \langle \Sigma, \Gamma, \Phi \rangle \\
\text{Pool map:} & \Phi & \in & (\ell : \mathit{Label}) \to \mathit{Pool}\ \ell
\end{array}
$$

Fig. 13: Decorated Calculus.

term *and* stack guarantees that the whole configuration represents a $MAC\ \ell$ computation.

It is known that dealing with dynamic allocation of memory makes it challenging to prove noninterference (e.g., [15, 53]). One manner to tackle this technicality is by establishing a bijection between public memory addresses of the two executions we want to relate and considering equality of public terms up to such notion [15]. Instead, and similar to other work [55, 145, 152, 153], we compartmentalize the memory into isolated labeled segments, one for each label of the lattice. This way, allocation in one segment does not affect the others. A similar argument holds for the heap and the thread pool, which we therefore also organize in partitions, accessed through the heap map Γ respectively the pool map Φ. Since we now have multiple heaps in one configuration, we need to annotate the *free* variables with the label of the heap in which they are bound. So a variable x^{ℓ} denotes that x is bound in the heap $\Gamma(\ell)$. Variables bound inside a term remain unlabeled, e.g., $\lambda x.x$. A variable $x^{\ell_{\mathrm{L}}}$ in a ℓ_{H}-labeled memory will have a label of at most the memory's sensitivity, $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. Unlike variables, we do not need to annotate memory cells n, as they only occur in a $\mathit{Ref}\ n$ expression, which carries a label in its type. So a reference $\mathit{Ref}\ n :: \mathit{Ref}\ \ell\ \tau$ points to the n-th entry in the ℓ-labeled memory. In the following, we write $\mathrm{fresh}(x^{\ell})$ to denote that variable x is fresh with respect to heap $\Gamma(\ell) = \Delta^{\ell}$ and stack S^{ℓ}. We write $\Gamma[\ell][x^{\ell}] := t$ for the heap map obtained by performing the update $\Gamma(\ell)[x^{\ell} \mapsto t]$, and likewise for stores and pool maps.

6.3 Decorated Semantics

The interesting rules of the annotated semantics are shown in Figure 14. The rules for the pure fragment of the calculus are adapted to work with labeled variables. Note that rule [$\textsc{App}_2$] replaces the bound, hence unlabeled, variable y with the labeled variable x^{ℓ} and thus maintains the invariant that *free* variables are labeled.

Why do we get away with giving the pure fragment of the annotated calculus only access to the heap $\Gamma(\ell)$ in a configuration at level ℓ? What if the program accesses a variable at a different level ℓ'? Because that cannot happen

$$\textsc{(App}_1\textsc{)}\quad \frac{\mathsf{fresh}(x^\ell)}{(\Delta^\ell, t_1\ t_2, S^\ell) \rightsquigarrow (\Delta^\ell[x^\ell \mapsto t_2], t_1, x^\ell : S^\ell)}$$

$$\textsc{(App}_2\textsc{)}\quad (\Delta^\ell, \lambda y.t, x^\ell : S^\ell) \rightsquigarrow (\Delta^\ell, t\ [x^\ell\ /\ y], S^\ell)$$

$$\textsc{(Var}_1\textsc{)}\quad (\Delta^\ell[x^\ell \mapsto t], x^\ell, S^\ell) \rightsquigarrow (\Delta^\ell, t, \#x^\ell : S^\ell)$$

$$\textsc{(Var}_2\textsc{)}\quad (\Delta^\ell, v, \#x^\ell : S) \rightsquigarrow (\Delta^\ell[x^\ell \mapsto v], v, S^\ell)$$

$$\textsc{(Lift)}\quad \frac{(\Gamma(\ell), t_1, S_1^\ell) \rightsquigarrow (\Delta^\ell, t_2, S_2^\ell)}{\langle \Sigma, \Gamma, t_1, S_1^\ell\rangle \longrightarrow \langle \Sigma, \Gamma[\ell \mapsto \Delta^\ell], t_2, S_2^\ell\rangle}$$

$$\textsc{(LazyDup}_1\textsc{)}\quad \frac{\neg\ \mathit{isVar}\ (t) \qquad \mathsf{fresh}(x^\ell)}{\langle \Sigma, \Gamma, \mathit{lazyDup}\ t, S^\ell\rangle \longrightarrow \langle \Sigma, \Gamma[\ell][x^\ell] := t, \mathit{lazyDup}\ x^\ell, S^\ell\rangle}$$

$$\textsc{(LazyDup}_2\textsc{)}\quad \frac{x^{\ell_L} \mapsto t\ \in\ \Sigma(\ell_L) \qquad \mathsf{fresh}(y^{\ell_H})}{\langle \Sigma, \Gamma, \mathit{lazyDup}\ x^{\ell_L}, S^{\ell_H}\rangle \longrightarrow \langle \Sigma, \Gamma[\ell_H][y^{\ell_H}] := [\![t]\!]^\varnothing, y^{\ell_H}, S^{\ell_H}\rangle}$$

$$\textsc{(New)}\quad \frac{|\Sigma(\ell_H)| = n \qquad \mathsf{fresh}(x^{\ell_L})}{\langle \Sigma, \Gamma, \mathit{new}\ t, S^{\ell_L}\rangle \longrightarrow \langle \Sigma[\ell_H][n] := x^{\ell_L}, \Delta[\ell_L][x^{\ell_L}] := t, \mathit{return}\ (\mathit{Ref}\ n), S^{\ell_L}\rangle}$$

$$\textsc{(Write}_2\textsc{)}\quad \frac{\mathsf{fresh}(x^{\ell_L})}{\langle \Sigma, \Gamma, \mathit{Ref}\ n, \mathit{write}\ t : S^{\ell_L}\rangle \longrightarrow \langle \Sigma[\ell_H][n] := x^{\ell_L}, \Gamma[\ell_L][x^{\ell_L}] := t, \mathit{return}\ (), S^{\ell_L}\rangle}$$

$$\textsc{(Read}_2\textsc{)}\quad \frac{\Sigma(\ell)[n] = x^\ell}{\langle \Sigma, \Gamma, \mathit{Ref}\ n, \mathit{read} : S^\ell\rangle \longrightarrow \langle \Sigma, \Gamma, \mathit{return}\ x^\ell, S^\ell\rangle}$$

$$\textsc{(Read}^D\textsc{)}\quad \langle \Sigma, \Gamma, \mathit{DRef}\ n, \mathit{read} : S^{\ell_H}\rangle \longrightarrow \langle \Sigma, \Gamma, \mathit{return}\ (\mathit{lazyDup}\ \Sigma(\ell_L)[n]), S^{\ell_H}\rangle$$

Fig. 14: Decorated Semantics

in a safe program, as the following example shows. Consider the following reduction sequence:

$$
\begin{array}{lllll}
 & ([x^{\ell'} \mapsto t], x^{\ell'}, & & [\,]) & \\
\rightsquigarrow & ([\,], & t, & \# x^{\ell'} : [\,]) & \text{-- rule [VAR}_1\text{]} \\
\rightsquigarrow^* & ([\,], & v, & \# x^{\ell'} : [\,]) & \\
\rightsquigarrow & ([x^{\ell'} \mapsto v], v, & & [\,]) & \text{-- rule [VAR}_2\text{]}
\end{array}
$$

In the first step, the ℓ-labeled configuration *reads* the variable $x^{\ell'}$. According to the *no read-up* security policy, this is only safe if $\ell' \sqsubseteq \ell$. In the last step, the ℓ'-labeled heap entry is updated with the value v. This constitutes a write operation, so accoring to the *no write-down* policy, this requires $\ell \sqsubseteq \ell'$. By the antisymmetry of the security lattice, it follows that $\ell \equiv \ell'$ must hold. So in the presence of *sharing*, a configuration complies with the *no write-down* and *no read-up* security policies only if it interacts soley with the ℓ-labeled heap.

Rule [LIFT] executes a pure reduction step, giving it access to the appropriate heap $\Gamma(\ell)$ and updating the heap map afterwards. Rules [LAZYDUP$_1$] and [LAZYDUP$_2$] adapt the semantics of *lazyDup* to label-partitioned heaps. The first rule takes care of allocating a non-trivial argument on the heap labeled as the current configuration. The second rule is the heart of our security leak fix: it handles the case where a high thread reads a thunk from a lower context. The rule fetches the thunk t from the lower heap, i.e., $t \mapsto \Sigma(\ell_{\text{L}})$, and extends the heap labeled as the configuration with a copy of the thunk, i.e., $\Sigma[\ell_{\text{H}}][y^{\ell_{\text{H}}}] := [\![t]\!]^{\varnothing}$. Observe that this operation relabels the original thunk t from ℓ_{L} to ℓ_{H} securely because t is duplicated, ensuring that the free variables in t will, by the time they are about to be evaluated, be wrapped in *lazyDup*, so that [LAZYDUP$_2$] kicks in again. In rule [NEW], a computation at level ℓ_{L} creates a reference labeled with ℓ_{H}. The thunk t is allocated on the ℓ_{L} heap under the name $x^{\ell_{\text{L}}}$, which is written to the fresh cell in memory $\Sigma(\ell_{\text{H}})$. This ensures the invariant that in *well-typed* configurations a memory holds references to lower heaps. The same applies to rule [WRITE$_2$]. Rule [READ$_2$] enforces that a computation at level ℓ can only read from a non-duplicated reference if the referenced variable is at the same level ℓ. Relaxing this would allow a high thread to read a thunk from a low level and thus open another leaky channel. But the interplay of rule [FORK] (in its annotated variant in Figure 16a in Appendix D), rule [LAZYDUP$_2$] and *lazyDup* rewriting of references to duplicate-on-read references precludes this scenario. Rule [READ^D] then allows a ℓ_{H} high computation to read a low variable from a duplicate-on-read reference, by duplicating it to ensure security.

6.4 Erasure Function

The term $\varepsilon_{\ell_A}(t :: \tau)$ is obtained from a term t with type τ by erasing data not observable by an attacker at level ℓ_A. For clarity, we omit the type annotation when irrelevant or obvious. Ground values (e.g., $()$, True) are unaffected by the

erasure function. For most syntactic forms, the function recurses homomorphically as in $\varepsilon_{\ell_A}(\mathit{lazyDup}\ t :: \tau) = \mathit{lazyDup}\ (\varepsilon_{\ell_A}(t :: \tau))$. The interesting cases are terms of type $\mathit{Labeled}\ \ell\ \tau$ and $\mathit{Ref}\ \ell\ \tau$. For such cases, the erasure function recurses as usual if $\ell \sqsubseteq \ell_A$. If, however, $\ell \not\sqsubseteq \ell_A$, and the resource is above the attacker's level, then it is erased and replaced by $\bullet$, e.g., $\varepsilon_{\ell_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ \ell\ \tau) = \mathit{Labeled}\ (\varepsilon_{\ell_A}(t :: \tau))$ if $\ell_A \sqsubseteq \ell$ or $\mathit{Labeled}\ \bullet$ otherwise. The erasure function is described with more detail in Appendix D.

6.5 Decorated Progress-Sensitive Non-interference

The noninterference proof relies on the two main properties *determinancy* and *simulation*. Determinancy simply states that transitions are deterministic:

Proposition 10 (Determinancy) *If $c_1 \hookrightarrow c_2$ and $c_1 \hookrightarrow c_3$ then $c_2 \equiv c_3$.*

The equality in this statement is alpha-equality, i.e., up to the choice of variables. In the machine-checked proofs all variables are De Bruijn indexes, and we indeed obtain structural equality.

The choice of determinism makes the concurrent model robust against scheduler refinement attacks. The second property, i.e., *simulation*, says that if a thread steps in a global configuration, then, either the same thread steps in the erased configuration, when the thread's level is visible to the attacker, i.e., $\ell \sqsubseteq \ell_A$, or otherwise, the initial and resulting configuration are indistinguishable to the attacker. We call such indistinguishability relation ℓ_A-equivalence, written $c_1 \approx_{\ell_A} c_2$ and defined as $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$. Observe that two ℓ_A-equivalent configurations contain exactly the same number of ℓ_A-equivalent public threads, but possibly a different number of secret threads. The notation $c_1 \hookrightarrow_{(\ell,n)} c_2$ expresses that the configuration c_1 runs the n-th thread at security level ℓ—threads are identified by label and number in the decorated semantics.

Proposition 11 (Simulation) *Given a global reduction step $c_1 \hookrightarrow_{(\ell,n)} c_1'$ then*

- *$\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c_1')$, if $\ell \sqsubseteq \ell_A$, or*
- *$c_1 \approx_{\ell_A} c_1'$, if $\ell \not\sqsubseteq \ell_A$*

From Propositions 10 and 11, we prove progress-sensitive noninterference. Note that, unlike our previous work [153], Proposition 11 does not simulate sensitive threads, because ℓ_A-equivalence suffices for PSNI. For more details, please refer to our Agda formalization[8].

Theorem 1 (PSNI) *Given two configurations c_1 and c_2, such that $c_1 \approx_{\ell_A} c_2$ and a reduction $c_1 \hookrightarrow_{(\ell,n)} c_1'$, then there exists a configuration c_2' such that $c_1' \approx_{\ell_A} c_2'$ and $c_2 \hookrightarrow^* c_2'$.*

As usual, $\hookrightarrow^*$ denotes the transitive reflexive closure of $\hookrightarrow$.

[8] Available at `https://github.com/marco-vassena/lazy-mac`

Proof 1 *We have two cases depending on whether the label ℓ is below or above the attacker's level.*
If $\ell \not\sqsubseteq \ell_A$, then we apply simulation (Proposition 11) to the step $c_1 \hookrightarrow_{(\ell,n)} c_1'$ and obtain $c_1 \approx_{\ell_A} c_1'$. Then, we pick $c_2' \equiv c_2$ ($c_2 \hookrightarrow^* c_2'$ in 0 steps) and derive $c_1' \approx_{\ell_A} c_2$ from symmetry and transitivity of the ℓ_A-equivalence relation applied to $c_1' \approx_{\ell_A} c_1$ and $c_1 \approx_{\ell_A} c_2$.
If $\ell \sqsubseteq \ell_A$, by ℓ_A-equivalence, configuration c_2 contains a thread identified by (ℓ, n), that is ℓ_A-equivalent to the thread (ℓ, n) run by c_1. However, configuration c_2 might contain a finite *number of high threads, which are scheduled before that. After running those high threads, i.e., $c_2 \hookrightarrow^* c_2''$, for some configuration c_2'', the same low thread is scheduled, i.e., $c_2'' \hookrightarrow_{(\ell,n)} c_2'$, for some other configuration c_2'. Then, we apply simulation (Proposition 11) to the first series of steps and obtain $c_2 \approx_{\ell_A} c_2''$ (they are all above the attackers level). From transitivity of the ℓ_A-equivalence relation applied to $c_1 \approx_{\ell_A} c_2$ and $c_2 \approx_{\ell_A} c_2''$, we then obtain $c_1 \approx_{\ell_A} c_2''$, i.e., $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2'')$. Then, we apply simulation again to the low steps ($\ell \sqsubseteq \ell_A$) and derive the corresponding erased step, i.e., $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c_1')$ from $c_1 \hookrightarrow_{(\ell,n)} c_1'$ and $\varepsilon_{\ell_A}(c_2'') \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c_2')$ from $c_2'' \hookrightarrow_{(\ell,n)} c_2'$. Lastly, we apply determinancy (Proposition 10) to the erased steps and obtain $\varepsilon_{\ell_A}(c_1') \equiv \varepsilon_{\ell_A}(c_2')$, which implies $c_1' \approx_{\ell_A} c_2'$.*

6.6 Simulation between Vanilla and Decorated semantics

To conclude the proofs of the security guarantees, we have to relate the decorated semantics with the vanilla semantics. On the one hand, we show that we can strip off the annotations from a decorated program, run it in the vanilla semantics, and get the same behaviour as running the decorated program in the decorated semantics. On the other hand, we show that we can annotate a well-typed vanilla program, based on the type derivations, and obtain a decorated program that executes correspondingly.

The main challenge is to map the partitioned heap, memory, and stack in the annotated calculus into a single heap, memory, and stack and vice versa. We apply techniques inspired by other IFC works on dynamic allocation [15] and partitioned heaps [55] and show that configurations in the annotated calculus are equal to those in the vanilla calculus *up to bijection on variables names and memory addresses*. These bijections describe how to flatten the partitioned memories and heaps into single entities without changing the results produced by programs—of course, modulo variable names and memory addresses. Note that our references are opaque to programs, i.e., there is no pointer arithmetic, equality, etc., which makes the proof easier.

We work with the two bijections, $\Psi_1 \in (Label \times Var) \to Var$ for heap variables and $\Psi_2 \in (Label \times \mathbb{N}) \to \mathbb{N}$ for memory addresses, where Var is the set of variables and $\mathbb{N}$ the set of memory addresses. When we refer to both bijections, we simply write Ψ. As one expects, we consider an annotated configuration *equivalent up to bjections* to a vanilla configuration, written $\langle \Sigma, \Gamma, t, S^\ell \rangle \cong_\Psi \langle M, \Delta, t', S \rangle$, if and only if their components are related, i.e., $\Sigma \cong_\Psi M$, $\Gamma \cong_\Psi \Delta$, $S^\ell \cong_\Psi S$, and $t \cong_\Psi t'$. The equiv-

alences on memories ($\Sigma \cong_\Psi M$), heap ($\Gamma \cong_\Psi \Delta$), and stack ($S^\ell \cong_\Psi S$) are defined point-wise. Equivalence of terms is a congruence relation with $x^\ell \cong_\Psi y$ if and only if $\Psi_1\ (\ell, x) = y$ and $Ref\ n :: Ref\ \ell\ \tau \cong_\Psi Ref\ m$ and $DRef\ n :: Ref\ \ell\ \tau \cong_\Psi DRef\ m$ if and only if $\Psi_2\ (\ell, n) = m$. Using this notion of equivalence modulo Ψ, we can state the simulation results:

Proposition 12 (Decorated to vanilla) *Given two well-typed configurations $\langle \Sigma, \Gamma, t_1, S^\ell \rangle$ and $\langle M, \Delta, t_2, S \rangle$ that denote a computation of type $MAC\ \ell\ \tau$, if we have that:*

- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \longrightarrow \langle \Sigma', \Gamma', t_1', S'^\ell \rangle$ *and*
- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \cong_\Psi \langle M, \Delta, t_2, S \rangle$

then there exist M', Δ', t_2', S' and Ψ' such that:

- $\langle M, \Delta, t_2, S \rangle \longrightarrow \langle M', \Delta', t_2', S' \rangle$
- $\langle \Sigma', \Gamma', t_1', S'^\ell \rangle \cong_{\Psi'} \langle M', \Delta', t_2', S' \rangle$

Note that the resulting configurations are in relation according to some new bijection Ψ', rather than Ψ, as the bijection has to be extended with new memory or heap allocations. Dually, we show that configurations in the vanilla calculus can be simulated in the annotated one.

Proposition 13 (Vanilla to decorated) *Given two well-typed configurations $\langle \Sigma, \Gamma, t_1, S^\ell \rangle$ and $\langle M, \Delta, t_2, S \rangle$ that denote a computation of type $MAC\ \ell\ \tau$, if we have that:*

- $\langle M, \Delta, t_2, S \rangle \longrightarrow \langle M', \Delta', t_2', S' \rangle$
- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \cong_\Psi \langle M, \Delta, t_2, S \rangle$

then there exist Σ', Γ', t_1', S'^ℓ and Ψ' such that:

- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \longrightarrow \langle \Sigma', \Gamma', t_1', S'^\ell \rangle$ *and*
- $\langle \Sigma', \Gamma', t_1', S'^\ell \rangle \cong_{\Psi'} \langle M', \Delta', t_2', S' \rangle$

We omit the typing rules, which are rather standard—the important bits are present in the type signatures given in Figures 2 and 7. For the decorated calculus, the typing rules correspond to those of the vanilla calculus, but in addition ensure that the security labels appearing in the type conincide with those in the decorations. The proof is rather standard including references and variables allocation, where we keep some invariant regarding the lengths of heap and memories to connect the notion of "freshness" of variables on both calculi. Detailed proofs of these simulations are in Appendix B.

6.7 Vanilla Progress-Sensitive Non-interference

We prove that *well-typed* programs in the vanilla lazy calculus satisfy progress-sensitive noninterference. This result relies on the PSNI proof for the decorated calculus and the simulations described above. We first define that two global configurations are ℓ_A-equivalence *up to a bijection* Ω, written $\langle M_1, \Delta_1, T_{s_1}\rangle \approx^{\Omega}_{\ell_A} \langle M_2, \Delta_2, T_{s_2}\rangle$, if and only if they are *well-typed* and their components are ℓ_A-equivalent up to bijection Ω, where ℓ_A-equivalence between terms is also type-driven and follows a structure similar to the one for the decorated calculus—the main difference being that it inspects the type-derivation of term and use the bijection Ω to relate memory addresses and heap variables. In the vanilla calculus we need to consider low-equivalence up to a bijection as in [15] to relate executions which might allocate a different amount of high entities, thus affecting the addresses and names of public references and variables respectively. Observe that bijection Ω connects heap variables and memory addresses of the vanilla calculus, that is Ω is a pair of bijections of type:

$$\Omega_1 :: Var \to Var$$
$$\Omega_2 :: \mathbb{N} \to \mathbb{N}$$

Configurations of the form $\langle [\,], \varnothing, [(t, [\,])]\rangle$ are initial configurations in the vanilla calculus, where the memory and thread's stacks are empty ($[\,]$), and the heap consists of an empty mapping ($\varnothing$).

Theorem 2 (Vanilla PSNI) *Given closed terms $t_1 :: MAC\ \ell\ \tau$ and $t_2 :: MAC\ \ell\ \tau$ written with the surface syntax (i.e., they do not contain constructors $Labeled$ and Ref), we have that if:*

- $t_1 \approx^{\varnothing}_{\ell_A} t_2$, *and*
- $\langle [\,], \varnothing, [(t_1, [\,])]\rangle \hookrightarrow^* c_1$, *then*

there exists c_2 and bijection Ω such that:

- $\langle [\,], \varnothing, [(t_2, [\,])]\rangle \hookrightarrow^* c_2$, *and*
- $c_1 \approx^{\Omega}_{\ell_A} c_2$

Proof 2 *(Sketch). Define $i_1 = \langle [\,], \varnothing, [(t_1, [\,])]\rangle$ and $i_2 = \langle [\,], \varnothing, [(t_2, [\,])]\rangle$. Since t_1 and t_2 are closed and well-typed terms in the surface syntax, we can lift them in the decorated calculus, as decorated terms t_1^D, t_2^D, and their corresponding initial annotated configurations i_1^D and i_2^D. Configurations i_1 and i_2 are equivalent up to the empty bijection $\varnothing$, i.e., $i_1^D \cong_{\emptyset} i_1$ and $i_2^D \cong_{\emptyset} i_2$ and $i_1^D \approx_{\ell_A} i_2^D$. By lifting Proposition 13 to thread pools and repetitively applying it, there exists a bijection Ψ_a and a configuration c_1^D, such that $i_1^D \hookrightarrow^* c_1^D$ and $c_1^D \cong_{\Psi_a} c_1$. By Theorem 1, there exists a decorated configuration c_2^D such that $i_2^D \hookrightarrow^* c_2^D$ and $c_1^D \approx_{\ell_A} c_2^D$. By lifting Proposition 12 to thread pools and repetitively applying it, we have that there exists a bijection Ψ_b and configuration c_2 such that $i_2 \hookrightarrow^* c_2$ where $c_2^D \cong_{\Psi_b} c_2$. We then conclude that c_1 and c_2 are ℓ_A-equivalent up to bijection Ω, obtained composing Ψ_a (from vanilla to decorated), and Ψ_b^{-1} (from decorated to vanilla), i.e., $c_1 \approx^{\Omega}_{\ell_A} c_2$, where $\Omega = \Psi_a \circ \Psi_b^{-1}$.*

7 Related Work

Mutable references and lazyness. In Section 3.2 we present an operational semantics that features both mutable references and lazyness. It is a straightforward combination of Sestoft's semantics with the standard approach to model references using a store, as described by Pierce et al. in the context of call-by-value [112, 113]. To the best of our knowledge, this is the first work that presents this combination. The "Awkward Squad" paper [110], which describes the implementation of I/O in Haskell, and addresses both references and concurrency, remarkably avoids dealing with sharing in its operational semantics.

deepDup. Our primitive *lazyDup* was inspired by the related primitive *deepDup* proposed by the second author [26], with the aim to limit sharing in cases where it is actually detrimental to program performance. Because the terms in that work are in Administrative Normal Form (ANF), the rules for *deepDup* look different from our [LazyDup$_2$], but this difference is inconsequential. We significantly improve over that work with the support to handle references, via the duplicate-on-read references introduced in Section 4.2. The Haskell library `ghc-dup` implements *deepDup* without changes to the compiler or runtime, therefore we are optimistic that an implementation of *lazyDup* is feasible.

Evaluation strategies and IFC. Sabelfeld and Sands suggest that lazy evaluation might be safer than eager evaluation for termination leaks [130]. Buiras and Russo identify the risk imposed by internal timing leaks via lazy evaluation [31]. Vassena et al. enrich **MAC**'s API for labeled expressions by considering them as (applicative-like) functors [152] and show that their extension is vulnerable to termination leaks under eager evaluation, but secure under lazy evaluation. In a imperative sequential setting, Rafnsson et al. describe how Java's on-demand (lazy) class initialization process can be exploit to reveal secrets [119]. Strictness analysis detects functions that always evaluate their arguments, which can then be eagerly evaluated to boost performance of lazy evaluation [98]. In this context, this technique could be used to safely force the evaluation of shared thunks upfront. However, the analysis must necessarily be conservative, especially when it comes to infinite data structures and advanced features such as references and concurrency, therefore it is unlikely that all leaks could be closed by the analysis alone. Nevertheless, strictness analysis could avoid unnecessary duplication: the thunks, which are guaranteed by the analysis to be evaluated anyway, could be eagerly forced, and lazy duplication could be applied otherwise.

IFC libraries. **LIO** dynamically enforces IFC applying similar concepts to **MAC** (i.e., labeled expressions, secure computations, etc.). We argue that **LIO** can be secure against the attack presented in this work by applying *lazyDup* to the "rest of the computation" every time that the current label gets raised. For that, **LIO** needs to be reimplemented to work in a continuation passing style (CPS)—we leave this direction as future work. **HLIO** (hybrid-**LIO**) works as **LIO**, except

it enforces IFC by combining type-level enforcement with dynamic checks [30]. To secure **HLIO**, *lazyDup* needs to be inserted when forking threads if IFC gets enforced statically and when raising the current label if dynamic checks are involved. **HLIO** also needs to be reimplemented using CPS. In **MAC**, the type signature for the bind operator restricts computations to maintain the same security level. Its type could be relaxed to involve different increasing labels, along the lines of the "is protected" relation used in the typing rule of bind in the Dependency Core Calculus (DCC) [1]. However, in that case, a secure computation would not enjoy a standard monadic structure, but it would rather incorporate multiple monads.

Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC [38]. Jaskelioff and Russo implement a library which dynamically enforces IFC using secure multi-execution (SME) [63]. Schmitz et al. provide a library with *faceted values*, where values present different behavior according to the privilege of the observer [132]. While these libraries do not support concurrency yet, we believe that, this work could secure them against lazy evaluation attacks, if they were extended with concurrency.

Programming languages. Besides the already mentioned tools Jif, Paragon, flowcaml, and JSFlow, we can remark the SPARK language and its IFC analysis, which has been extended to guarantee progress-sensitive non-inference [117] and JOANA [139], which stretches the scalability of static analyzes, in this case of Java programs. Some tools apply dependent-types to protect confidentiality (e.g., [82, 94, 102]). In such languages, type-checking triggers evaluation, potentially opening up possibilities to leak sensitive data via covert channels (e.g., lazy evaluation). In this light, it would be possible to learn something about a static secret when type-checking the program—an interesting direction for future work. Laminar combines programming languages and operating systems techniques to provide decentralized information flow control [122]. While supporting concurrency, Laminar does not handle covert channels like termination or internal timing leaks.

8 Conclusions

We present a solution to internal timing leaks via lazy evaluation, an open problem for security libraries written in Haskell. We believe that repairing existing libraries with *lazyDup* would be reasonably a painless experience. The utilization of *lazyDup* would make past and future systems built with security libraries more secure (e.g., Hails [45]). Even though it is still not clear which evaluation strategy is more beneficial for security, this work shows that the risks of lazy evaluation in concurrent settings can be successfully avoided.

Generally speaking, functional languages (and Haskell in particular) rely on their runtime (e.g., lazy evaluation, garbage collector, etc.) to provide essential features. Unfortunately, besides providing their functionality, they could also be misused to jeopardize security. This work shows that a program can control parts of the complex runtime system (e.g., sharing) via a safe interface

(*lazyDup*). Then, the obvious question is which other features of the runtime system could jeopardize security and how to safely control them—an intriguing thought to drive our future work.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM.
2. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.
3. Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 162–173, New York, NY, USA, 2000. ACM.
4. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, March 2005.
5. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
6. Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*. ACM, 2007.
7. Joachim Breitner. dup – explicit un-sharing in haskell. *CoRR*, abs/1207.2017, 2012.
8. Joachim Breitner. Formally proving a compiler transformation safe. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 35–46, New York, NY, USA, 2015. ACM.
9. Niklas Broberg, Bart Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 217–232, Berlin, Heidelberg, 2013. Springer-Verlag.
10. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.
11. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proceedings of the 18th Nordic Conference on Secure IT Systems - Volume 8208*, NordSec 2013, pages 116–122, New York, NY, USA, 2013. Springer-Verlag New York, Inc.
12. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communication of the ACM*, 20(7):504–513, July 1977.
13. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.
14. Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.
15. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web

applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

16. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing*. ACM, 2014.
17. Daniel Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.
18. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 11–31, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
19. J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.
20. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, pages 170–178, Berlin, Heidelberg, 2012. Springer-Verlag.
21. John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM.
22. Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 16–, Washington, DC, USA, 2006. IEEE Computer Society.
23. Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, April 2010.
24. Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328, New York, NY, USA, 2015. ACM.
25. Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.
26. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
27. Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.
28. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming*, pages 269–281, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.
29. Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.
30. Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, SP. IEEE Computer Society, 2011.
31. Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

32. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
33. Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.
34. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
35. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. http://www.cis.upenn.edu/ bcpierce/sf.
36. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.
37. Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. Progress-sensitive security for spark. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, pages 20–37, Berlin, Heidelberg, 2016. Springer-Verlag.
38. Willard Rafnsson, Keiko Nakata, and Andrei Sabelfeld. Securing class initialization in Java-like languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1), January 2013.
39. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2009.
40. Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 280–288, New York, NY, USA, 2015. ACM.
41. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.
42. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, Mar 2001.
43. Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca Viganò, editors, *POST*, volume 9635 of *LNCS*. Springer, 2016.
44. Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, May 1997.
45. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.
46. Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using JOANA. *it - Information Technology*, 56(6):280–287, 2014.
47. Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.

48. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.
49. Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 538–557, 2016.
50. Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 15–28, New York, NY, USA, 2016. ACM.
51. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.
52. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 563–574, New York, NY, USA, 2011. ACM.
53. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *ACM Conference on Programming Language Design and Implementation*. ACM, 2012.

$$\textsc{Var}\quad \frac{x \in Dom(\Gamma(\ell)) \qquad y \in Dom(\Delta) \qquad \Psi_{\Gamma,\Delta}(\ell, x) = y \qquad \Psi^{-1}_{\Gamma,\Delta}(y) = (\ell, x)}{x^{\ell} \cong_{\Psi_{\Gamma,\Delta}} y}$$

$$\textsc{Heap}\quad \frac{\forall\ \ell\ x\ y \qquad x^{\ell} \cong_{\Psi_{\Gamma,\Delta}} y \qquad \Gamma(\ell)(x) \cong_{\Psi_{\Gamma,\Delta}} \Delta(y)}{\Gamma \cong_{\Psi_{\Gamma,\Delta}} \Delta}$$

$$\textsc{Ref}\quad \frac{n < |\Sigma(\ell)| \qquad m < |\Delta| \qquad \Psi_{\Sigma,M}(\ell, n) = m \qquad \Psi^{-1}_{\Sigma,M}(m) = (\ell, n)}{Ref_{\ell}\ n \cong_{\Psi_{\Sigma,M}} Ref\ m}$$

$$\textsc{Memory}\quad \frac{\forall\ \ell\ n\ m \qquad Ref_{\ell}\ n \cong_{\Psi_{\Sigma,M}} Ref\ m \qquad \Sigma(\ell)[n] \cong_{\Psi_{\Sigma,M}} M[m]}{\Sigma \cong_{\Psi_{\Sigma,M}} M}$$

$$\textsc{Stack}_1\quad [\,]^{\ell} \cong_{\Psi} [\,]$$

$$\textsc{Stack}_2\quad \frac{C^{\ell} \cong_{\Psi} C \qquad S^{\ell} \cong_{\Psi} S}{C^{\ell} : S^{\ell} \cong_{\Psi} C : S}$$

$$\textsc{Var}_{\#}\quad \frac{x \notin Dom(\Gamma(\ell)) \qquad y \notin Dom(\Delta)}{\#x^{\ell} \cong_{\Psi_{\Gamma,\Delta}} \#y}$$

Fig. 15: Definition of $\cong_{\Psi_{\Gamma,\Delta}}$ and $\cong_{\Psi_{\Sigma,M}}$.

Appendix

A Securing LIO

In **LIO**, it is not possible to know, at the time of forking, if the parent or the spawned thread will become sensitive, because threads get dynamically "tainted" when they observe a piece of sensitive information, e.g., by means of *unlabel*—an approach known as *floating-label system*. One could follow the same idea used in **MAC** and conservatively apply *lazyDup* to all spawned threads. However, such approach would overly restrict sharing, e.g., if the thread never observes secrets. Instead, *lazyDup* should be applied to the "rest of the computation" whenever the thread gets tainted—only then the evaluation of thunks can leak information! Implementing this idea requires to refactor the full implementation of **LIO** to work in a continuation-passing style, where the continuation represents the "rest of the computation". Then, when the thread gets tainted, *lazyDup* can be applied to the continuation, thus disabling sharing with the parent thread from that point on.

B Simulation Proof

In this section, we prove *simulation* between the decorated and vanilla calculi (i.e., Propositions 12 and 13 from Section 6) for the interesting cases, i.e., rules [$\textsc{Var}_1$,$\textsc{Var}_2$,$\textsc{LazyDup}_1$, $\textsc{LazyDup}_2$,$\textsc{New}$]. Firstly, we refine the type for *bijection* on heap variables with a heap map Γ and a heap Δ. In particular we write Var^{Δ} to restrict the type of heap variables to those in the domain of Δ, i.e.,

$Var^{\Delta} = \{ x \mid x \in Dom(\Delta) \}$. Similarly, we write $\mathbb{N}^M$, to restrict the type of memory addresses to those in the domain of memory M, i.e., $\mathbb{N}^M = \{ n \mid n < |M| \}$. We then write $(\ell : Label \times P(\ell))$, for the dependent pair type (also known as Sigma-type) $\Sigma_{(\ell:Label)} P(\ell)$. We then give the following more precise type to heap variables and memory addresses bijections:

$$\Psi_{\Gamma,\Delta} :: (\ell : Label \times Var^{\Gamma(\ell)}) \to Var^{\Delta}$$
$$\Psi_{\Sigma,M} :: (\ell : Label \times \mathbb{N}^{\Sigma(\ell)}) \to \mathbb{N}^{M}$$

In particular heap-indexed and memory-indexed bijections relate *only* variables and addresses in their domains. In the following we abbreviate the pair of bijection $(\Psi_{\Gamma,\Delta}, \Psi_{\Sigma,M})$ with Ψ, and sometimes we specify only the relevant component of the pair to avoid clutter. Figure 15 shows the definition of equivalence up to heap-bijection ($\cong_{\Psi_{\Gamma,\Delta}}$) and equivalence up to memory-bijection ($\cong_{\Psi_{\Sigma,M}}$) for the interesting cases. Rule [Var] relates the variables in the domain of $\Gamma(\ell)$ and Δ respectively, using the bijection $\Psi_{\Gamma,\Delta}$. Rule [Heap] defines equivalence of heaps up to bijection pointwise for the variables in their domain, i.e., a store and a heap are equivalent up to bijection, if and only if they map related variables into related terms. Rules [Ref,Memory] apply the same principles to memory addresses. In decorated calculus we write $Ref_{\ell}\ n$, to denote that the reference has type $Ref\ \ell\ \tau$, for some type τ—the vanilla reference $Ref\ m$ has the same type. Note that related stacks share the same structure, i.e., all their continuations are related, where the only interesting case involve the update marker continuation, i.e., $\#x$. Rule [Var$_{\#}$] states that a decorated continuation $\#x^{\ell}$ and a vanilla continuation $\#y$ are related if and only if *both* variables are *free* in their respective heaps. Term-equivalence up to bijection is defined inductively on their structures, e.g., $lazyDup\ t^D \cong_{\Psi} lazyDup\ t$ if and only if $t^D \cong_{\Psi} t$. We remark that these relations are defined over well-typed terms of the *same* type, that is $t^D \cong_{\Psi} t$, assumes typing judgment $\pi \vdash t : \tau$, for some typing context π, and that t^D has type τ in the same typing context—we distinguish decorated terms (e.g., t^D), from vanilla terms (e.g., t), with a superscript. The typing rules for the vanilla calculus are standard and thus omitted—they corresponds to the type signatures given in Figures 2 and 7.

Weakening. If two configurations are equivalent up to bijection Ψ, i.e., $c^D \cong_{\Psi} c$, then they are equivalent up to any bijection $\Psi \cup \{x^{\ell} \leftrightarrow y\}$, for any pair of variables x^{ℓ} and y, that are *fresh* in the respective configurations.

Strengthening. If two configurations are equivalent via an extended bijection, e.g., $c^D \cong_{\Psi \cup \{x^{\ell} \leftrightarrow y\}} c$, then they are also equivalent in a reduced bijection, e.g., $c^D \cong_{\Psi} c$, if and only if they occur in their respective stack under an update marker, e.g., $\#x^{\ell}$ and $\#y$.

We conclude with the proofs of the simulation propositions between the annotated and vanilla calculi (Propositions 12 and 13) for rules [Var$_1$,Var$_2$,LazyDup$_1$, LazyDup$_2$, New].

- Rule [VAR$_1$].
  - *Decorated to Vanilla*: Given a decorated step $(\Delta^\ell[x^\ell \mapsto t^D], x^\ell, S^\ell) \rightsquigarrow (\Delta^\ell, t^D, \#x^\ell : S^\ell)$, a vanilla configuration (Δ, t, S) and a bijection Ψ, such that $(\Delta^\ell[x^\ell \mapsto t^D], x^\ell, S^\ell) \cong_\Psi (\Delta, t, S)$, show that there exists a configuration (Δ', t', S') such that $(\Delta, t, S) \rightsquigarrow (\Delta', t', S')$ and a bijection Ψ' such that $(\Delta^\ell, t, \#x^\ell : S^\ell) \cong_{\Psi'} (\Delta', t', S')$. Since the initial configurations are in relation, then so are their heaps (recall $\Delta^\ell = \Sigma(\ell)$), current terms and stacks. Therefore the term t is a variable y, such that $x^\ell \cong_{\Psi_{\Gamma,\Delta} \cup \{x^\ell \leftrightarrow y\}} y$ and the heap Δ contains a binding for y, that is $\Delta^\ell[x^\ell \mapsto t^D] \cong_{\Psi_{\Gamma,\Delta} \cup \{x^\ell \leftrightarrow y\}} \Delta[y \mapsto t]$. The vanilla configuration then steps according to rule [VAR$_1$], i.e., $(\Delta[y \mapsto t], y, S) \rightsquigarrow (\Delta, t, \#y : S)$, which is equivalent to the decorated configuration up to the bijection $\Psi_{\Gamma,\Delta}$, i.e., the bijection obtained by removing mapping $x^\ell \leftrightarrow y$. Note that $t^D \cong_\Psi t$, since they are the image of related variables in related heaps and $\#x^\ell : S^\ell \cong_{\Psi_{\Gamma,\Delta}} \#y : S$, since the stacks are related and the variables are *both* free in Γ and Δ respectively—rule [VAR$_1$] removes them to achieve the *blackholing* effect.
  - *Vanilla to Decorated*: The proof is symmetric. In this case we have a vanilla [VAR$_1$] step, i.e., $(\Delta[y \mapsto t], y, S) \rightsquigarrow (\Delta, t, \#y : S)$. Since the vanilla configuration denotes a secure computation of type $MAC\ \ell\ \tau$, for some label ℓ and some type τ, then the equivalent decorated configuration is labeled with ℓ, i.e., $(\Delta^\ell[x^\ell \mapsto t^D], x^\ell, S^\ell)$. The proof then follows similarly, by making the same considerations about the vanilla configuration to draw the same conclusions about the decorated configuration.
- Rule [VAR$_2$].
  - *Vanilla to Decorated*: Consider a vanilla step [VAR$_2$], i.e., $(\Delta, v, \#x : S) \rightsquigarrow (\Delta[x \mapsto v], v, S)$. Since the configuration denotes a secure computation of type $MAC\ \ell\ \tau$, the Ψ-equivalent decorated configuration is labeled with ℓ and has the same shape, i.e., $(\Delta^\ell, v^D, \#x^\ell : S^\ell)$—if a vanilla term v is a value then v^D is also a value and if the top of a vanilla stack has an update marker, so does the equivalent decorated stack, by rules [STACK$_2$, VAR$_\#$]. The decorated configuration then steps according to rule [VAR$_2$], to the configuration $(\Delta^\ell[x^\ell \mapsto v^D], v^D, S^\ell)$, which is equivalent to the vanilla configuration up to bijection $\Psi \cup \{x^\ell \leftrightarrow x\}$—we extend related heaps with related terms, i.e., $v^D \cong_\Psi v$. Note that from $\#x^\ell \cong_{\Psi_{\Gamma,\Delta}} \#x$, we have that x^ℓ and x are *free* in $\Delta^\ell = \Gamma(\ell)$ and Δ respectively. By popping related continuations from related stacks we obtain related stacks, i.e., $S^\ell \cong_\Psi S$.
  - *Decorated to Vanilla*: The proof follows symmetrically. In this case the label ℓ is explicitly available in the decorated configuration.
- Rule [LAZYDUP$_1$].
  - *Vanilla to Decorated*: Given a vanilla step $\langle M, \Delta, lazyDup\ t, S\rangle \longrightarrow \langle M, \Delta[x \mapsto t], lazyDup\ x, S\rangle$, i.e., rule [LAZYDUP$_2$], lifted by rule

[LIFT], where x is fresh in Δ and t is not a variable. Observe that, since the configuration denotes a secure computation of type $MAC\ \ell\ \tau$ for some label ℓ and some type τ, then the equivalent decorated initial configuration is labeled with ℓ, and it has $lazyDup\ \ t^D$ as the current term, where $t^D \cong_\Psi t$ and $\neg\ (isVar\ \ t^D)$. Then, the decorated configuration $\langle \Sigma, \Gamma, lazyDup\ \ t^D, S^\ell \rangle$ steps according to rule [LAZYDUP$_1$] to $\langle \Sigma, \Gamma[\ell][x^\ell] := t^D, lazyDup\ \ x^\ell, S^\ell \rangle$, for some fresh variable x^ℓ in $\Gamma(\ell)$. The final configurations are then equivalent up to the bijection $\Psi \cup \{x^\ell \leftrightarrow x\}$—note that this is a bijection because x^ℓ and x are fresh in $\Gamma(\ell)$ and Δ respectively, hence they are not mapped in Ψ. Specifically the heaps are extended with related terms and hence are related by the extended bijection $\Psi \cup \{x^\ell \leftrightarrow x\}$, and $lazyDup\ t^D \cong_{\Psi \cup \{x^\ell \leftrightarrow x\}} lazyDup\ t$.

- *Decorated to Vanilla*: The proof follows symmetrically. In this case the label ℓ is explicitly available in the decorated configuration and the step is simulated in the vanilla calculus by rule [LAZYDUP$_1$] lifted to vanilla sequential configuration by rule [LIFT].

– Rule [LAZYDUP$_2$].

- *Vanilla to Decorated* Given a vanilla step $\langle M, \Delta, lazyDup\ \ x, S \rangle \longrightarrow \langle M, \Delta[y \mapsto [\![t]\!]^\varnothing], y, S \rangle$, i.e., rule [LAZYDUP$_2$], lifted by rule [LIFT], where variable y is fresh and variable x is bound to term t in the heap Δ, the Ψ-equivalent decorated configuration contains a Ψ-equivalent heap map Γ, i.e. $\Gamma \cong_\Psi \Delta$, a Ψ-equivalent stack S^{ℓ_H}, i.e. $S^{\ell_H} \cong_\Psi S$, and a Ψ-equivalent current term $lazyDup\ \ x^{\ell_L}$, i.e. $lazyDup\ \ x^{\ell_L} \cong_\Psi lazyDup\ \ x$, from which it follows that $x^{\ell_L} \cong_\Psi x$. Since variables and heaps are related, we have that the corresponding thunks are also related, i.e. there exists t^D, such that $\Gamma(\ell_L)(x^{\ell_L}) = t^D$ and $t^D \cong_\Psi t$. The decorated configuration then steps according to rule [LAZYDUP$_2$], giving heap map $\Gamma[\ell_H][y^{\ell_H}] := [\![t^D]\!]^\varnothing$ and current term y^{ℓ_H}, for some fresh variable y^{ℓ_H}. The resulting decorated configuration is then equivalent to the vanilla configuration up to the bijection $\Psi' = \Psi \cup \{y^{\ell_H} \leftrightarrow y\}$—it is a bijection because the variables are fresh. The heaps are related, i.e. $\Gamma[\ell_H][y^{\ell_H}] := [\![t^D]\!]^\varnothing \cong_{\Psi'} \Delta[y \mapsto [\![t]\!]^\varnothing]$, because we extend related heaps with related terms—function $[\![]\!]^\varnothing$ preserves equivalence up to bijection, i.e. if $t^D \cong_\Psi t$ then $[\![t^D]\!]^\varnothing \cong_\Psi [\![t]\!]^\varnothing$. The current terms are related by definition, i.e. $y^{\ell_H} \cong_{\Psi'} y$, because $(y^{\ell_H} \leftrightarrow y) \in \Psi'$.
- *Decorated to Vanilla*: The proof follows symmetrically. In this case the label ℓ is explicitly available in the decorated configuration and the step is simulated in the vanilla calculus by rule [LAZYDUP$_2$] lifted to vanilla sequential configuration by rule [LIFT].

– Rule [NEW]

- *Vanilla to Decorated*: Consider a vanilla step $\langle M[n \mapsto x], \Delta[x \mapsto t], return\ \ (Ref\ \ n), S \rangle$, where $|M| = n$ and x is fresh. The Ψ-equivalent configuration, consists of a Ψ-equivalent store Σ, i.e., $\Sigma \cong_\Psi$

M, a Ψ-equivalent heap map Γ, i.e., $\Gamma \cong_{\Psi} \Delta$, a Ψ-equivalent current term $new\ t^D$, i.e., $new\ t^D \cong_{\Psi} new\ t$, from which we have $t^D \cong_{\Psi} t$. From the type derivation of the *well-typed* vanilla configuration, we know that term $new\ t$ has type $MAC\ \ell_{\mathrm{L}}\ (Ref\ \ell_{\mathrm{H}}\ \tau)$, for some type τ and some labels ℓ_{L} and ℓ_{H}, such that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. The decorated configuration steps according to rule [New], giving for a fresh variable $x^{\ell_{\mathrm{L}}}$ and $m = |\Sigma(\ell_{\mathrm{H}})|$, the store $\Sigma[\ell_{\mathrm{H}}][m] := x^{\ell_{\mathrm{L}}}$, heap map $\Delta[\ell_{\mathrm{L}}][x^{\ell_{\mathrm{L}}}] := t^D$ and current term $return\ (Ref_{\ell_{\mathrm{L}}}\ m)$. The resulting configurations are equivalent up to the bijection $\Psi' = (\Psi_{\Gamma,\Delta} \cup \{x^{\ell_{\mathrm{L}}} \leftrightarrow x\}, \Psi_{\Sigma,M} \cup \{(\ell_{\mathrm{H}}, m) \leftrightarrow n\})$, i.e., the bijection obtained by extending the heap variables and memory addresses bijections with the new mappings $x^{\ell_{\mathrm{L}}} \leftrightarrow x$ and $(\ell_{\mathrm{H}}, m) \leftrightarrow n$ respectively. The heaps are equivalent up to the bijection $\Psi_{\Gamma,\Delta} \cup \{x^{\ell_{\mathrm{L}}} \leftrightarrow x\}$ since we extend $\Psi_{\Gamma,\Delta}$-equivalent heaps variables with respectively *fresh* variables $x^{\ell_{\mathrm{L}}}$ and x, which are bound to Ψ-equivalent terms, i.e., $t^D \cong_{\Psi} t$. Similarly, the memories are equivalent up to the bijection $\Psi_{\Sigma,M} \cup \{(\ell_{\mathrm{H}}, m) \leftrightarrow n\}$, since we extend $\Psi_{\Sigma,M}$-equivalent memories, by assigning equivalent references i.e., $x^{\ell_{\mathrm{L}}} \cong_{\Psi_{\Gamma,\Delta} \cup \{x^{\ell_{\mathrm{L}}} \leftrightarrow x\}} x$ to *fresh* addresses. Observe that the current terms in the final configurations are related, i.e., $return\ (Ref_{\ell_{\mathrm{L}}}\ m) \cong_{\Psi'} return\ (Ref\ n)$, because the addresses are related by the bijection Ψ', i.e., $m \cong_{\Psi_{\Sigma,M} \cup \{(\ell_{\mathrm{H}}, m) \leftrightarrow n\}} n$.

- *Decorated to Vanilla*: The proof follows symmetrically. In this case the labels ℓ_{L} and ℓ_{H} are explicitly available in the decorated configurations, i.e., $\langle \Sigma, \Gamma, new\ t^D, S^{\ell_{\mathrm{L}}} \rangle \rightsquigarrow \langle \Sigma[\ell_{\mathrm{H}}][m] := x^{\ell_{\mathrm{L}}}, \Gamma[\ell_{\mathrm{L}}][x^{\ell_{\mathrm{L}}}] := t^D, return\ (Ref_{\ell_{\mathrm{H}}}\ m), S^{\ell_{\mathrm{L}}} \rangle$.

C Sharing and References

Our calculus captures sharing precisely, even in presence of references, and despite the extra-indirection between the memory and heap. We provide two examples showing the interaction among references, sharing, and thunks.

Example 1. Consider the following program, which creates a reference, immediately overwrites it with 1, and finally returns 0:

$$
\begin{array}{l}
\textbf{let}\ x = 0\ \textbf{in} \\
\quad \textbf{do}\ r \leftarrow new\ x \\
\qquad write\ r\ 1 \\
\qquad return\ x
\end{array}
$$

If reference r pointed directly to x (no extra-indirection), the next write operation would actually rewrite x to 1 in the *immutable* heap and the program would return 1, instead of 0.

Example 2. Consider the following program, which writes a thunk in a reference, reads it and evaluates its content twice.

$$
\begin{array}{l}
\textbf{let } x = id\ 1 \textbf{ in} \\
\quad \textbf{do } r \leftarrow new\ x \\
\qquad\ \ y \leftarrow read\ r \\
\qquad\ \ \textbf{when } (y \leqslant 0)\ return\ () \\
\qquad\ \ z \leftarrow read\ r \\
\qquad\ \ \textbf{when } (z \geqslant 0)\ return\ ()
\end{array}
$$

This program demands the value of y to evaluate $y \leqslant 0$ and the value to z to evaluate $z \geqslant 0$, but, surprisingly enough, the value of z is already computed. This sounds counter-intuitive because we expect y and z to be bound to the same expression $id\ 1$, since the program does not overwrite reference r between the first and the second read. In fact, variables y and z are *aliases* of the same variable x, whose thunk $id\ 1$ is updated with 1 after checking $y \leqslant 0$, thanks to sharing, and used to check $z \geqslant 0$. Observe that, while this program does not contain an explicit *write* operation, it still does perform one subtly, in the heap, since it *indirectly* updates x.

D Erasure Function

Figure 16b shows the definition of the erasure functions for the interesting cases. Configurations, whose label is above that of the attacker, i.e., $\ell \not\sqsubseteq \ell_A$ are rewritten to $\bullet$, otherwise they are erased by erasing each component. Steps involving sensitive configurations are then simulated by rules [Hole$_1$, Hole$_2$] from Figure 16a. Memories, heaps, stacks and thread pools labeled with ℓ are also collapsed to $\bullet$, if their label is not visible to the attacker, i.e., $\ell \not\sqsubseteq \ell_A$, otherwise they are erased homomorphically. Label partitioned data structures, i.e., heap maps, stores and pool maps, are erased pointwise, e.g. $\varepsilon_{\ell_A}(\Gamma) = \ell \mapsto \varepsilon_{\ell_A}(\Gamma(\ell))$. The term $\mathit{label}\ \ t :: \mathit{MAC}\ \ \ell_{\mathrm{L}}\ \ (\mathit{Labeled}\ \ \ell_{\mathrm{H}}\ \ \tau)$ is erased to $\mathit{label}\ \bullet$, if $\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$, so that rule [Label] commutes. The terms new, write, fork are interesting. Observe that all these terms perform a *write-effect*, to a non-lower security level, due to the no write-down policy, which allows a computation visible to the attacker ($\ell_{\mathrm{L}} \sqsubseteq \ell_A$) to write to a non-visible resource ($\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$). Simulating such steps, i.e., the label-decorated version of rules [New, Write, Fork], is challenging and requires *two-steps erasure* [153], a technique that performs erasure in two-stages, by firstly rewriting the problematic constructs, such as new, write and fork to special constructs, i.e., $\mathit{new}_\bullet$, $\mathit{write}_\bullet$ and $\mathit{fork}_\bullet$, whose special semantics rule guarantees simulation. We remark that such special constructs are introduced due to mere technical reasons and they are not part of the plain calculus. We use $\mathit{fork}_\bullet$ as an example to illustrate this technique. Figure 16a shows rules [Fork] and [Fork$_\bullet$], that is the label annotated rules for fork and $\mathit{fork}_\bullet$ respectively. Rule [Fork] is similar to its annotated counterpart shown in Figure 11, save for the extra look-up and update through the thread pool map Φ. Rule [Fork$_\bullet$] mimics rule [Fork] for what concerns the parent thread, but it ignores thread t, which is not added to the thread pool. Observe that rule [Fork] does not correctly simulate fork operations that occur in high threads. In particular, the high thread pool T_{s_2} is rewritten by the erasure function to $\bullet$, since $\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$, however $|\bullet| \not\equiv m$.

On the other hand by rewriting fork to a new term, i.e., $\mathit{fork}_\bullet$, we are free to adjust its semantics, to correctly simulate a low thread forking a high one in erased configurations. Specifically, we can show that [Fork] commutes with [Fork$_\bullet$], by proving that for all thread pool maps Φ, Φ', such that $\Phi' = \Phi[\ell_{\mathrm{H}} \mapsto (t, S^{\ell_{\mathrm{H}}})]$ and $\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$, then $\varepsilon_{\ell_A}(\Phi) \equiv \varepsilon_{\ell_A}(\Phi')$, i.e., the attacker is oblivious to writes in thread pools above its security level.

$$\text{(Hole}_1\text{)} \quad \bullet \rightsquigarrow \bullet \qquad\qquad \text{(Hole}_2\text{)} \quad \bullet \longrightarrow \bullet$$

(Fork)

$$\frac{\Phi(\ell_{\mathrm{L}}) = T_{s_1}[n \mapsto (\mathit{fork}\ t, S^{\ell_{\mathrm{L}}})] \qquad \Phi(\ell_{\mathrm{H}}) = T_{s_2} \qquad |T_{s_2}| = m \qquad T'_{s_1} = T_{s_1}[n \mapsto (\mathit{return}\ (), S^{\ell_{\mathrm{L}}})] \qquad T'_{s_2} = T_{s_2}[m \mapsto (\mathit{lazyDup}\ t, [\,]^{\ell_{\mathrm{H}}})]}{\langle \Sigma, \Gamma, \Phi \rangle \hookrightarrow \langle \Sigma, \Gamma, \Phi[\ell_{\mathrm{L}} \mapsto T'_{s_1}][\ell_{\mathrm{H}} \mapsto T'_{s_2}] \rangle}$$

(Fork$_\bullet$)

$$\frac{\Phi(\ell_{\mathrm{L}}) = T_s[n \mapsto (\mathit{fork}_\bullet\ t, S^{\ell_{\mathrm{L}}})] \qquad T'_s = T_s[n \mapsto (\mathit{return}\ (), S^{\ell_{\mathrm{L}}})]}{\langle \Sigma, \Gamma, \Phi \rangle \hookrightarrow \langle \Sigma, \Gamma, \Phi[\ell_{\mathrm{L}} \mapsto T'_s] \rangle}$$

(a) Semantics rules for extended calculus.

$$\varepsilon_{\ell_A}(\langle \Delta^\ell, t, S^\ell \rangle) = \begin{cases} \langle \varepsilon_{\ell_A}(\Delta^\ell), \varepsilon_{\ell_A}(t), \varepsilon_{\ell_A}(S^\ell) \rangle & \text{if } \ell \sqsubseteq \ell_A \\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\langle \Sigma, \Gamma, t, S^\ell \rangle) = \begin{cases} \langle \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Gamma), \varepsilon_{\ell_A}(t), \varepsilon_{\ell_A}(S^\ell) \rangle & \text{if } \ell \sqsubseteq \ell_A \\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{Ref}\ t :: \mathit{Ref}\ \ell_{\mathrm{H}}\ n) = \begin{cases} \mathit{Ref}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{Ref}\ n & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{label}\ t :: \mathit{Mac}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} \mathit{label}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{label}\ (\varepsilon_{\ell_A}(t :: \tau)) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{fork}\ t :: \mathit{Mac}\ \ell_{\mathrm{L}}\ ()) = \begin{cases} \mathit{fork}_\bullet\ (\varepsilon_{\ell_A}(t :: \mathit{Mac}\ \ell_{\mathrm{H}}\ ())) & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{fork}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\bullet) = \bullet$$

(b) Erasure function.

Fig. 16: Extended calculus and erasure function.

Paper V

Based on

From Fine- to Coarse-Grained Dynamic Information Flow Control and Back,

by Marco Vassena, Alejandro Russo, Deepak Garg,

Vineet Rajani and Deian Stefan,

46th ACM SIGPLAN Symposium on Principles of Programming Languages.

CHAPTER

FOUR

FROM FINE- TO COARSE-GRAINED DYNAMIC INFORMATION FLOW CONTROL AND BACK

Abstract. We show that fine-grained and coarse-grained dynamic information flow control (IFC) systems are equally expressive. To this end, we mechanize two mostly standard languages, one with a fine-grained dynamic IFC system and the other with a coarse-grained dynamic IFC system, and prove a semantics-preserving translation from each language to the other. In addition, we derive the standard security property of non-interference of each language from that of the other, via our verified translation. This result addresses a longstanding open problem in IFC: whether coarse-grained dynamic IFC techniques are less expressive than fine-grained dynamic IFC techniques (they are not!). The translations also stand to have important implications on the usability of IFC approaches. The coarse- to fine-grained direction can be used to remove the label annotation burden that fine-grained systems impose on developers, while the fine- to coarse-grained translation shows that coarse-grained systems—which are easier to design and implement—can track information as precisely as fine-grained systems and provides an algorithm for automatically retrofitting legacy applications to run on existing coarse-grained systems.

1 Introduction

Dynamic *information-flow control* (IFC) is a principled approach to protecting the confidentiality and integrity of data in software systems. Conceptually, dynamic IFC systems are very simple—they associate *security* levels or *labels* with every bit of data in the system to subsequently track and restrict the flow of labeled data throughout the system, e.g., to enforce a security property such as *non-interference* [47]. In practice, dynamic IFC implementations are considerably more complex—the *granularity* of the tracking system alone has important implications for the usage of IFC technology. Indeed, until somewhat

recently [122, 144], granularity was the main distinguishing factor between dynamic IFC operating systems and programming languages. Most IFC operating systems (e.g., [39, 69, 166]) are *coarse-grained*, i.e., they track and enforce information flow at the granularity of a process or thread. Conversely, most programming languages with dynamic IFC (e.g., [7, 51, 59, 164, 165]) track the flow of information in a more *fine-grained* fashion, e.g., at the granularity of program variables and references.

Dynamic coarse-grained IFC systems in the style of LIO [30, 55, 123, 141, 144, 145] have several advantages over dynamic fine-grained IFC systems. Such coarse-grained systems are often easier to design and implement—they inherently track less information. For example, LIO protects against control-flow-based *implicit flows* by tracking information at a coarse-grained level—to branch on secrets, LIO programs must first taint the context where secrets are going to be observed. Finally, coarse-grained systems often require considerably fewer programmer annotations—unlike fine-grained ones. More specifically, developers often only need a single label-annotation to protect everything in the scope of a thread or process responsible to handle sensitive data.

Unfortunately, these advantages of coarse-grained systems give up on the many benefits of fine-grained ones. For instance, one main drawback of coarse-grained systems is that it requires developers to compartmentalize their application in order to avoid both false alarms and the *label creep* problem, i.e., wherein the program gets too "tainted" to do anything useful. To this end, fine-grained systems often create special abstractions (e.g., event processes [39], gates [166], and security regions [122]) that compensate for the conservative approximations of the coarse-grained tracking approach. Furthermore, fine-grained systems do not impose the burden of focusing on avoiding the label creep problem on developers. By tracking information at fine granularity, such systems are seemingly more flexible and do not suffer from false alarms and label creep issues [7] as coarse-grained systems do. Indeed, fine-grained systems such as JSFlow [51] can often be used to secure existing, legacy applications; they only require developers to properly annotate the application.

This paper removes the division between fine- and coarse-grained dynamic IFC systems and the belief that they are fundamentally different. In particular, we show that *dynamic* fine-grained and coarse-grained IFC are equally expressive. Our work is inspired by the recent work of [120, 121], who prove similar results for *static* fine-grained and coarse-grained IFC systems. Specifically, they establish a semantics- and type-preserving translation from a coarse-grained IFC type system to a fine-grained one and vice-versa. We complete the picture by showing a similar result for dynamic IFC systems that additionally allow *introspection on labels* at run-time. While label introspection is meaningless in a static IFC system, in a dynamic IFC system this feature is key to both writing practical applications and mitigating the label creep problem [144].

Using Agda, we formalize a traditional fine-grained system (in the style of [7]) extended with label introspection primitives, as well as a coarse-grained

system (in the style of [144]). We then define and formalize modular semantics-preserving translations between them. Our translations are macro-expressible in the sense of [40].

We show that a translation from fine- to coarse-grained is possible when the coarse-grained system is equipped with a primitive that limits the scope of tainting (e.g., when reading sensitive data). In practice, this is not an imposing requirement since most coarse-grained systems rely on such primitives for compartmentalization. For example, [141, 144], provide `toLabeled` blocks and threads for precisely this purpose. Dually, we show that the translation from coarse- to fine-grained is possible when the fine-grained system has a primitive $\mathbf{taint}(\cdot)$ that relaxes precision to keep the *program counter label* synchronized when translating a program to the coarse-grained language. While this primitive is largely necessary for us to establish the coarse- to fine-grained translation, extending existing fine-grained systems with it is both secure and trivial.

The implications of our results are multi-fold. The fine- to coarse-grained translation formally confirms an old OS-community hypothesis that it is possible to restructure a system into smaller compartments to address the label creep problem—indeed our translation is a (naive) algorithm for doing so. This translation also allows running legacy fine-grained IFC compatible applications atop coarse-grained systems like LIO. Dually, the coarse- to fine-grained translation allows developers building new applications in a fine-grained system to avoid the annotation burden of the fine-grained system by writing some of the code in the coarse-grained system and compiling it automatically to the fine-grained system with our translation. The technical contributions of this paper are:

- A pair of semantics-preserving translations between traditional dynamic fine-grained and coarse-grained IFC systems equipped with label introspection (Theorems 3 and 5).
- Two different proofs of *termination-insensitive* non-interference (TINI) for each calculus: one is derived directly in the usual way (Theorems 1 and 2), while the other is recovered via our verified translation (Theorems 4 and 6).
- Mechanized Agda proofs of our results (~4,000 LOC).[1]

The rest of this paper is organized as follows. Our dynamic fine- and coarse-grained IFC calculi are introduced in Sections 2 and 3, respectively. We also prove their soundness guarantees (i.e., termination-insensitive non-interference). Section 4 presents the translation from the fine- to the coarse-grained calculus and recovers the non-interference of the former from the non-interference theorem of the latter. Section 5 has similar results in the other direction. Related work is described in Section 6 and Section 7 concludes the paper.

[1] Artifact available at `https://hub.docker.com/r/marcovassena/granularity/`

Type:	$\tau ::=$	$\mathbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathscr{L} \mid \mathbf{Ref}\ \tau$
Labels:	$\ell, pc \in$	$\mathscr{L}$
Address:	$n \in$	$\mathbb{N}$
Environment:	$\theta \in$	$Var \rightharpoonup Value$
Raw Value:	$r ::=$	$() \mid (x.e, \theta) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid (v_1, v_2) \mid \ell \mid n_\ell$
Value	$v ::=$	r^ℓ
Expression:	$e ::=$	$x \mid \lambda x.e \mid e_1\ e_2 \mid () \mid \ell \mid \mathbf{inl}(e) \mid \mathbf{inr}(e)$
		$\mid \mathbf{case}(e, x.e_1, x.e_2) \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e)$
		$\mid \mathbf{getLabel} \mid \mathbf{labelOf}(e) \mid \mathbf{taint}(e_1, e_2) \mid \mathbf{new}(e)$
		$\mid !\, e \mid e_1 := e_2 \mid \mathbf{labelOfRef}(e) \mid e_1 \sqsubseteq^? e_2$
Type System:	$\Gamma \vdash e : \tau$	
Configuration:	$c ::=$	$\langle \Sigma, e \rangle$
Store:	$\Sigma \in$	$(\ell : Label) \to Memory\ \ell$
Memory ℓ:	$M ::=$	$[] \mid r : M$

Fig. 1: Syntax of λ^{dFG}.

2 Fine-Grained Calculus

In order to compare in a rigorous way fine- and coarse-grained dynamic IFC techniques, we formally define the operational semantics of two λ-calculi that respectively perform fine- and coarse-grained IFC dynamically. Figure 1 shows the syntax of the dynamic fine-grained IFC calculus λ^{dFG}, which is inspired by [7] and extended with a standard (security unaware) type system $\Gamma \vdash e : \tau$ (omitted), sum and product data types and security labels $\ell \in \mathscr{L}$ that form a lattice $(\mathscr{L}, \sqsubseteq)$.[2] In order to capture flows of information precisely at run-time, the λ^{dFG}-calculus features *intrinsically labeled* values, written r^ℓ, meaning that raw value r has security level ℓ. Compound values, e.g., pairs and sums, carry labels to tag the security level of each component, for example a pair containing a secret and a public boolean would be written $(\mathbf{true}^H, \mathbf{false}^L)$.[3] Functional values are closures $(x.e, \theta)$, where x is the variable that binds the argument in the body of the function e and all other free variables are mapped to some labeled value in the environment θ. The λ^{dFG}-calculus features a labeled partitioned stored, i.e., $\Sigma \in (\ell : \mathscr{L}) \to Memory\ \ell$, where $Memory\ \ell$ is the memory that contains values at security level ℓ. Each reference carries an additional label annotation that records the label of the memory it refers to—reference n_ℓ points to the n-th cell of the ℓ-labeled memory, i.e., $\Sigma(\ell)$. Notice that this label has nothing to do with the *intrinsic* label that decorates the reference itself. For example, a reference $(n_H)^L$ represents a secret reference in a public context, whereas $(n_L)^H$ represents a public reference in a secret context. Notice that there is no order invariant between those labels—in the latter case, the IFC runtime monitor prevents writing data to the reference to

[2] The lattice is arbitrary and fixed. In examples we will often use the two point lattice $\{L, H\}$, which only disallows secret to public flow of information, i.e., $H \not\sqsubseteq L$.

[3] We define the boolean type $\mathbf{bool} = \mathbf{unit} + \mathbf{unit}$, boolean values as raw values, i.e., $\mathbf{true} = \mathbf{inl}(()^L)$, $\mathbf{false} = \mathbf{inr}(()^L)$ and $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 = \mathbf{case}\ e\ \_.e_1\ \_.e_2$.

$$\textsc{(Var)}\quad \langle \Sigma, x\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, \theta(x) \sqcup pc\rangle \qquad \textsc{(Unit)}\quad \langle \Sigma, ()\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, ()^{pc}\rangle \qquad \textsc{(Label)}\quad \langle \Sigma, \ell\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, \ell^{pc}\rangle$$

$$\textsc{(Fun)}\quad \langle \Sigma, \lambda x.e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, (x.e, \theta)^{pc}\rangle$$

$$\textsc{(App)}\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', (x.e, \theta')^{\ell}\rangle \quad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v_2\rangle \quad \langle \Sigma'', e\rangle \Downarrow^{\theta'[x\mapsto v_2]}_{pc\sqcup\ell} \langle \Sigma''', v\rangle}{\langle \Sigma, e_1\ e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma''', v\rangle}$$

$$\textsc{(Inl)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v\rangle}{\langle \Sigma, \mathbf{inl}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inl}(v)^{pc}\rangle} \qquad \textsc{(Inr)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v\rangle}{\langle \Sigma, \mathbf{inr}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inr}(v)^{pc}\rangle}$$

$$\textsc{(Case}_1\textsc{)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inl}(v_1)^{\ell}\rangle \quad \langle \Sigma', e_1\rangle \Downarrow^{\theta[x\mapsto v_1]}_{pc\sqcup\ell} \langle \Sigma'', v\rangle}{\langle \Sigma, \mathbf{case}(e, x.e_1, x.e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v\rangle}$$

$$\textsc{(Case}_2\textsc{)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inr}(v_2)^{\ell}\rangle \quad \langle \Sigma', e_2\rangle \Downarrow^{\theta[x\mapsto v_2]}_{pc\sqcup\ell} \langle \Sigma'', v\rangle}{\langle \Sigma, \mathbf{case}(e, x.e_1, x.e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v\rangle}$$

$$\textsc{(Pair)}\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v_1\rangle \quad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v_2\rangle}{\langle \Sigma, (e_1, e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', (v_1, v_2)^{pc}\rangle}$$

$$\textsc{(Fst)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', (v_1, v_2)^{\ell}\rangle}{\langle \Sigma, \mathbf{fst}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v_1 \sqcup \ell\rangle} \qquad \textsc{(Snd)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', (v_1, v_2)^{\ell}\rangle}{\langle \Sigma, \mathbf{snd}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v_2 \sqcup \ell\rangle}$$

$$\textsc{(Taint)}\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell^{\ell'}\rangle \quad \ell' \sqsubseteq \ell \quad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{\ell} \langle \Sigma'', v\rangle}{\langle \Sigma, \mathbf{taint}(e_1, e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v\rangle}$$

Fig. 2: Big-step semantics for λ^{dFG} (part I).

avoid *implicit flows*. A program can create, read and write a labeled reference via constructs $\mathbf{new}(e)$, $!e$ and $e_1 := e_2$ and inspect its subscripted label with the primitive $\mathbf{labelOfRef}(\cdot)$.

2.1 Dynamics

The operational semantics of λ^{dFG} includes a security monitor that propagates the label annotations of input values during program execution and assigns

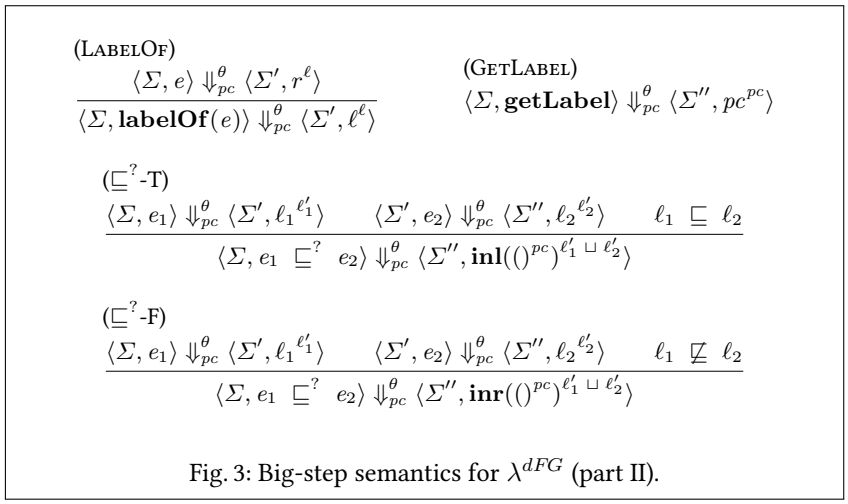


Fig. 3: Big-step semantics for λ^{dFG} (part II).

security labels to the result accordingly. The monitor prevents information leakage by stopping the execution of potentially leaky programs, which is reflected in the semantics by not providing reduction rules for the cases that may cause insecure information flow.[4] The relation $\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v\rangle$ denotes the evaluation of program e with initial store Σ that terminates with labeled value v and final store Σ'. The environment θ stores the input values of the program and is extended with intermediate results during function application and case analysis. The subscript pc is the *program counter* label [128]— it is a label that represents the security level of the context in which the expression is evaluated. The semantics employs the program counter label to (i) propagate and assign labels to values computed by a program and (ii) prevent implicit flow leaks that exploit the control flow and the store (explained below).

In particular, when a program produces a value, the monitor tags the raw value with the program counter label in order to record the security level of the context in which it was computed. For this reason all the introduction rules for ground and compound types ([Unit,Label,Fun,Inl,Inr,Pair]) assign security level pc to the result. Other than that, these rules are fairly standard—we simply note that rule [Fun] creates a closure by capturing the current environment θ.

When the control flow of a program *depends* on some intermediate value, the program counter label is joined with the value's label so that the label of the final result will be tainted with the result of the intermediate value. For instance, consider case analysis, i.e., $\mathbf{case}\ e\ x.e_1\ x.e_2$. Rules [Case$_1$] and [Case$_2$] evaluate the scrutinee e to a value (either $\mathbf{inl}(v)^{\ell}$ or $\mathbf{inr}(v)^{\ell}$), add the value to the environment, i.e., $\theta[x \mapsto v]$, and then execute the appropriate branch with a program counter label tainted with v's security label, i.e., $pc \sqcup \ell$. As a result, the monitor tracks data dependencies across control flow constructs through

[4] In this work, we ignore leaks that exploit program termination and prove *termination insensitive* non-interference for λ^{dFG} (Theorem 1).

the label of the result. Function application follows the same principle. In rule [App], since the first premise evaluates the function to some closure $(x.e, \theta')$ at security level ℓ, the third premise evaluates the body with program counter label raised to $pc \sqcup \ell$. The evaluation strategy is call-by-value: it evaluates the argument before the body in the second premise and binds the corresponding variable to its value in the environment of the closure, i.e., $\theta'[x \mapsto v_2]$. Notice that the security level of the argument is irrelevant at this stage and that this is beneficial to not over-tainting the result: if the function never uses its argument then the label of the result depends exclusively on the program counter label, e.g., $(\lambda x.())\ y \Downarrow_L^{y \mapsto 42^H} ()^L$. The elimination rules for variables and pairs taint the label of the corresponding value with the program counter label for security reasons. In rules [Var,Fst,Snd] the notation, $v \sqcup \ell'$ upgrades the label of v with ℓ'—it is a shorthand for $r^{\ell \sqcup \ell'}$ with $v = r^{\ell}$. Intuitively, public values must be considered secret when the program counter is secret, for example $x \Downarrow_H^{x \mapsto ()^L} ()^H$.

Label Introspection. The λ^{dFG}-calculus features primitives for label introspection, namely $\mathbf{getLabel}$, $\mathbf{labelOf}(\cdot)$ and $\sqsubseteq^?$—see Figure 3. These operations allow to respectively retrieve the current program counter label, obtain the label annotations of values, and compare two labels (inspecting labels at run-time is useful for controlling and mitigating the label creep problem).

Enabling label introspection raises the question of what label should be assigned to the label itself (in λ^{dFG} every value, including all label values, must be annotated with a label). As a matter of fact, labels can be used to encode secret information and thus careless label introspection may open the doors to information leakage [144]. Notice that in λ^{dFG}, the label annotation on the result is computed by the semantics together with the result and thus it is as sensitive as the result itself (the label annotation on a value depends on the sensitivity of all values affecting the *control-flow* of the program up to the point where the result is computed). This motivates the design choice to protect each projected label with the label itself, i.e., ℓ^{ℓ} and pc^{pc} in rules [GetLabel] and [LabelOf] in Figure 2. We remark that this choice is consistent with previous work on coarse-grained IFC languages [32, 144], but novel in the context of fine grained IFC.

Finally, primitive $\mathbf{taint}(e_1, e_2)$ temporarily raises the program counter label to the label given by the first argument in order to evaluate the second argument. The fine-to-coarse translation in Section 4 uses $\mathbf{taint}(\cdot)$ to loosen the precision of λ^{dFG} in a controlled way and match the *coarse* approximation of our coarse-grained IFC calculus (λ^{dCG}) by upgrading the labels of intermediate values systematically. In rule [Taint], the constraint $\ell' \sqsubseteq \ell$ ensures that the label of the nested context ℓ is at least as sensitive as the program counter label pc. In particular, this constraint ensures that the operational semantics have Property 1 (*"the label of the result is at least as sensitive as the program counter label"*) even with rule [Taint].

(New)
$$\frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell}\rangle \qquad n = |\Sigma'(\ell)|}{\langle \Sigma, \mathbf{new}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'[\ell \mapsto \Sigma'(\ell)[n \mapsto r]], (n_{\ell})^{pc}\rangle}$$

(Read)
$$\frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {n_{\ell}}^{\ell'}\rangle \qquad \Sigma'(\ell)[n] = r}{\langle \Sigma, !e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell \sqcup \ell'}\rangle}$$

(Write)
$$\frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {n_{\ell}}^{\ell_1}\rangle \qquad \ell_1 \sqsubseteq \ell \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', r^{\ell_2}\rangle \qquad \ell_2 \sqsubseteq \ell}{\langle \Sigma, e_1 := e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma''[\ell \mapsto \Sigma''(\ell)[n \mapsto r]], {}^{pc}\rangle}$$

(LabelOfRef)
$$\frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {n_{\ell}}^{\ell'}\rangle}{\langle \Sigma, \mathbf{labelOfRef}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell^{\ell \sqcup \ell'}\rangle}$$

Fig. 4: Big-step semantics for λ^{dFG} (references).

Property 1 *If* $\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell}\rangle$ *then* $pc \sqsubseteq \ell$.

Proof. By induction on the given evaluation derivation.

References. We now extend the semantics presented earlier with primitives that inspect, access and modify the labeled store via labeled references. See Figure 4. Rule [New] creates a reference n_{ℓ}, labeled with the security level of the initial content, i.e., label ℓ, in the ℓ-labeled memory $\Sigma(\ell)$ and updates the memory store accordingly.[5] Since the security level of the reference is as sensitive as the content, which is at least as sensitive as the program counter label by Property 1 ($pc \sqsubseteq \ell$) this operation does not leak information via *implicit flows*. When reading the content of reference n_{ℓ} at security level ℓ', rule [Read] retrieves the corresponding raw value from the n-th cell of the ℓ-labeled memory, i.e., $\Sigma'(\ell)[n] = r$ and upgrades its label to $\ell \sqcup \ell'$ since the decision to read from that particular reference depends on information at security level ℓ'. When writing to a reference the monitor performs security checks to avoid leaks via explicit or implicit flows. Rule [Write] achieves this by evaluating the reference, i.e., ${(n_{\ell})}^{\ell_1}$ and replacing its content with the value of the second argument, i.e., r^{ℓ_2}, under the conditions that the decision of "which" reference to update does not depend on data more sensitive than the reference itself, i.e., $\ell_1 \sqsubseteq \ell$ (not checking this would leak via an *implicit flow*)[6], and that the new content is no more sensitive than the reference itself, i.e., $\ell_2 \sqsubseteq \ell$ (not

[5] $|M|$ denotes the length of memory M—memory indices start at 0.

[6] Notice that $pc \sqsubseteq \ell_1$ by Property 1, thus $pc \sqsubseteq \ell_1 \sqsubseteq \ell$ by transitivity. An *implicit flow* would occur if a reference is updated in a *high branch*, i.e., depending on the secret, e.g., **let** $x = \mathbf{new}(0)$ **in if** $secret$ **then** $x := 1$ **else** ().

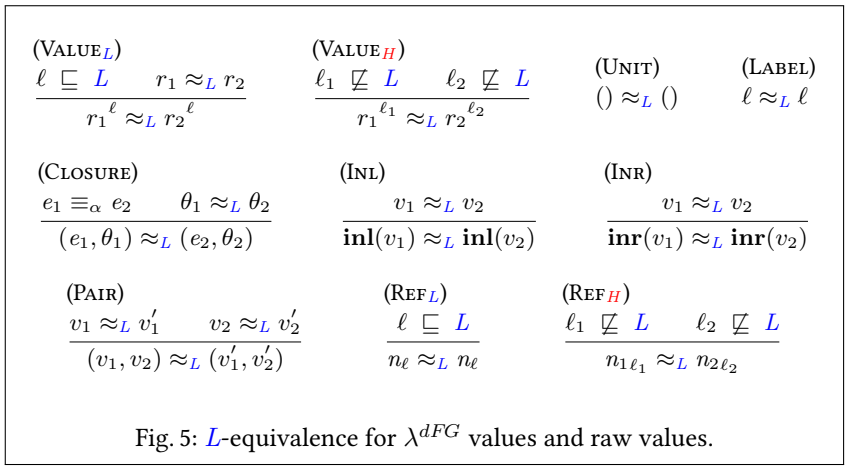


Fig. 5: L-equivalence for λ^{dFG} values and raw values.

checking this would leak sensitive information to a less sensitive reference via an *explicit flow*). Lastly, rule [LABELOFREF] retrieves the label of the reference and protects it with the label itself (as explained before) and taints it with the security level of the reference, i.e., $\ell^{\ell \sqcup \ell'}$ to avoid leaks. Intuitively, the label of the reference, i.e., ℓ, depends also on data at security level ℓ' as seen in the premise.

Other Extensions. We consider λ^{dFG} equipped with references as sufficient foundation to study the relationship between fine-grained and coarse-grained IFC. We remark that extending it with other side-effects such as file operations, or other IO-operations would not change our claims in Section 4 and 5. The main reason for this is that, typically, handling such effects would be done at the same granularity in both IFC enforcements. For instance, when adding file operations, both fine- (e.g., [28]) and coarse-grained (e.g., [39, 69, 124, 145]) enforcements are likely to assign a single *flow-insensitive* label to each file in order to denote the sensitivity of its content. Then, those features could be handled *flow-insensitively* in both systems (e.g., [100, 116, 145, 153]), in a manner similar to what we have just shown for references in λ^{dFG}.

2.2 Security

We now prove that λ^{dFG} is secure, i.e., it satisfies *termination insensitive non-interference* (TINI) [47, 158]. Intuitively, the security condition says that no terminating λ^{dFG} program leaks information, i.e., changing secret inputs does not produce any publicly visible effect. The proof technique is standard and based on the notion of L-equivalence, written $v_1 \approx_L v_2$, which relates values (and similarly raw values, environments, stores and configurations) that are indistinguishable for an attacker at security level L. For clarity we use the 2-points lattice, assume that secret data is labeled with H and that the attacker can only observe data at security level L. Our mechanized proofs are parametric in the lattice and in the security level of the attacker. L-equivalence for values and

raw-values is defined formally by mutual induction in Figure 5. Rule [VALUE$_L$] relates observable values, i.e., raw values labeled below the security level of the attacker. These values have the *same* observable label ($\ell \sqsubseteq L$) and related raw values, i.e., $r_1 \approx_L r_2$. Rule [VALUE$_H$] relates non-observable values, which may have different labels not below the attacker level, i.e., $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$. In this case, the raw values can be arbitrary. Raw values are L-equivalent when they consist of the same ground value ([UNIT,LABEL]), or are homomorphically related for compound values. For example, for the sum type the relation requires that both values are either a left or a right injection ([INL,INR]). In particular, closures are related if they contain the *same* function (up to α-renaming)[7] and L-equivalent environments, i.e., the environments are L-equivalent pointwise. Formally, $\theta_1 \approx_L \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x.\theta_1(x) \approx_L \theta_2(x)$.

We define L-equivalence for stores pointwise, i.e., $\Sigma_1 \approx_L \Sigma_2$ iff for all labels $\ell \in \mathscr{L}$, $\Sigma_1(\ell) \approx_L \Sigma_2(\ell)$. Memory L-equivalence relates arbitrary ℓ-labeled memories if $\ell \not\sqsubseteq L$, and pointwise otherwise, i.e., $M_1 \approx_L M_2$ iff M_1 and M_2 are memories labeled with $\ell \sqsubseteq L$, $|M_1| = |M_2|$ and for all $n \in \{0\,..\,|M_1|-1\}$, $M_1[n] \approx_L M_2[n]$. Similarly, L-equivalence relates any two secret references (rule [REF$_H$]) but requires the same label and address for public references (rule [REF$_L$]). We naturally lift L-equivalence to initial configurations, i.e., $c_1 \approx_L c_2$ iff $c_1 = \langle \Sigma_1, e_1 \rangle$, $c_2 = \langle \Sigma_2, e_2 \rangle$, $\Sigma_1 \approx_L \Sigma_2$ and $e_1 \equiv_\alpha e_2$, and final configurations, i.e., $c_1' \approx_L c_2'$ iff $c_1' = \langle \Sigma_1', v_1 \rangle$, $c_2' = \langle \Sigma_2', v_2 \rangle$ and $\Sigma_1' \approx_L \Sigma_2'$ and $v_1 \approx_L v_2$.
We now formally state and prove that λ^{dFG} semantics preserves L-equivalence of configurations under L-equivalent environments, i.e., *termination-insensitive non-interference* (TINI).

Theorem 1 (λ^{dFG}-TINI)
If $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ *and* $c_1 \approx_L c_2$ *then* $c_1' \approx_L c_2'$.

Proof. By induction on the derivations.

Dynamic language-based fine-grained IFC, of which λ^{dFG} is just a particular instance, represents an intuitive approach to tracking information flows in programs. Programmers annotate input values with labels that represent their sensitivity and a label-aware instrumented security monitor propagates those labels during execution and computes the result of the program together with a conservative approximation of its sensitivity. The next section describes an IFC monitor that tracks information flows at *coarse* granularity.

3 Coarse-Grained Calculus

One of the drawbacks of dynamic fine-grained IFC is that the programming model requires all input values to be explicitly and fully annotated with their

[7] Symbol $\equiv_\alpha$ denotes α-equivalence. In our mechanized proofs we use De Bruijn indexes and syntactic equivalence.

Type:	τ ::=	$\mathbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathscr{L}$ $\mid \mathbf{LIO}\ \tau \mid \mathbf{Labeled}\ \tau \mid \mathbf{Ref}\ \tau$
Labels:	$\ell, pc \in$	$\mathscr{L}$
Address:	$n \in$	$\mathbb{N}$
Environment:	$\theta \in$	$Var \rightharpoonup Value$
Value:	v ::=	$() \mid (x.e, \theta) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid (v_1, v_2) \mid \ell$ $\mid (t, \theta) \mid \mathbf{Labeled}\ \ell\ v \mid n_\ell$
Expression:	e ::=	$x \mid \lambda x.e \mid e_1\ e_2 \mid () \mid \ell \mid e_1 \sqsubseteq^? e_2 \mid \mathbf{inl}(e_1) \mid \mathbf{inr}(e_2)$ $\mid \mathbf{case}(e, x.e_1, x.e_2) \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid t$
Thunk	t ::=	$\mathbf{return}(e) \mid \mathbf{bind}(e, x.e) \mid \mathbf{unlabel}(e)$ $\mid \mathbf{toLabeled}(e) \mid \mathbf{labelOf}(e) \mid \mathbf{getLabel} \mid \mathbf{taint}(e)$ $\mid \mathbf{new}(e) \mid !\, e \mid e_1 := e_2 \mid \mathbf{labelOfRef}(e)$
Type System:	$\Gamma \vdash e : \tau$	
Configuration:	c ::=	$\langle \Sigma, pc, e \rangle$
Store:	$\Sigma \in$	$(\ell : Label) \to Memory\ \ell$
Memory ℓ:	M ::=	$[\,] \mid v : M$

Fig. 6: Syntax of λ^{dCG}.

security labels. Imagine a program with many inputs and highly structured data: it quickly becomes cumbersome, if not impossible, for the programmer to specify all the labels. The label of some inputs may be sensitive (e.g., passwords, pin codes, etc.), but the sensitivity of the rest may probably be irrelevant for the computation, yet a programmer must come up with appropriate labels for them as well. The programmer is then torn between two opposing risks: over-approximating the actual sensitivity can negatively affect execution (the monitor might stop secure programs), under-approximating the sensitivity can endanger security. Even worse, specifying many labels manually is error-prone and assigning the wrong security label to a piece of sensitive data can be catastrophic for security and completely defeat the purpose of IFC. Dynamic coarse-grained IFC represents an attractive alternative that requires fewer annotations, in particular it allows the programmer to label only the inputs that need to be protected.

Figure 6 shows the syntax of λ^{dCG}, a standard simply-typed λ-calculus extended with security primitives for dynamic coarse-grained IFC, inspired by [145] and adapted to use call-by-value instead of call-by-name to match λ^{dFG}. The λ^{dCG}-calculus features both standard (unlabeled) values and *explicitly labeled* values. For example, $\mathbf{Labeled}\ H\ \mathbf{true}$ represents a secret boolean value of type $\mathbf{Labeled\ bool}$.[8] The type constructor $\mathbf{LIO}$ encapsulates a security state monad, whose state consists of a labeled store and the program counter label. In addition to standard $\mathbf{return}(\cdot)$ and $\mathbf{bind}(\cdot)$ constructs, the monad provides primitives that regulate the creation and the inspection of

[8] As in λ^{dFG}, we define $\mathbf{bool} = \mathbf{unit} + \mathbf{unit}$ and $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 = \mathbf{case}\ e\ \_.e_1\ \_.e_2$. Unlike λ^{dFG} values, λ^{dCG} values are not intrinsically labeled, thus we encode boolean constants simply as $\mathbf{true} = \mathbf{inl}()$ and $\mathbf{false} = \mathbf{inr}()$.

labeled values, i.e., $\mathbf{toLabeled}(\cdot)$, $\mathbf{unlabel}(\cdot)$ and $\mathbf{labelOf}(\cdot)$, and the interaction with the labeled store, allowing the creation, reading and writing of labeled references n_ℓ through the constructs $\mathbf{new}(e)$, $!e$, $e_1 := e_2$, respectively. The primitives of the **LIO** monad are listed in a separate sub-category of expressions called *thunk*. Intuitively, a thunk is just a description of a stateful computation, which only the top-level security monitor can execute—a *thunk closure*, i.e., (t, θ), provides a way to suspend computations.

3.1 Dynamics

In order to track information flows dynamically at coarse granularity, λ^{dCG} employs a technique called *floating-label*, which was originally developed for IFC operating systems (e.g., [166, 167]) and that was later applied in a language-based setting. In this technique, throughout a program's execution, the program counter *floats* above the label of any value observed during program execution and thus represents (an upper-bound on) the sensitivity of all the values that are not explicitly labeled. For this reason, λ^{dCG} stores the program counter label in the program configuration, so that the primitives of the **LIO** monad can control it explicitly (in technical terms the program counter is *flow-sensitive*, i.e., it may assume different values in the final configuration depending on the control flow of the program).[9]

Like λ^{dFG}, the operational semantics of λ^{dCG} consists of a security monitor that fully evaluates secure programs but prevents the execution of insecure programs and similarly enforces *termination-insensitive* non-interference (Theorem 2). Figure 7 shows the big-step operational semantics of λ^{dCG} in two parts: (i) a top-level security monitor for monadic programs and (ii) a straightforward call-by-value side-effect-free semantics for pure expressions. The semantics of the security monitor is further split into two mutually recursive reduction relations, one for arbitrary expressions (Fig. 7a) and one specific to thunks (Fig. 7c). These constitute the *forcing* semantics of the monad, which reduce a thunk to a pure value and perform side-effects. In particular, given the initial store Σ, program counter label pc, expression e of type $\mathbf{LIO}\ \tau$ for some type τ and input values θ (which may or may not be labeled), the monitor executes the program, i.e., $\langle \Sigma, pc, e\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$ and gives an updated store Σ', updated program counter pc' and a final value v of type τ, which also might not be labeled. The execution starts with rule [Force], which reduces the pure expression to a thunk closure, i.e., (t, θ') and then forces the thunk t in its environment θ' with the thunk semantics. The pure semantics is fairly standard—we report some selected rules in Fig. 7b for comparison with λ^{dFG}. A pure reduction, written $e \Downarrow^{\theta} v$, evaluates an expression e with an appropriate environment θ to a pure value v. Notice that, unlike λ^{dFG}, all reduction rules of the pure semantics ignore security, even those that affect the control flow of the program, e.g., rule [App]: they do not feature the program counter label or label

[9] In contrast, we consider λ^{dFG}'s program counter *flow-insensitive* because it is part of the evaluation judgment and its value changes only inside nested judgments.

(Force)
$$\frac{e \Downarrow^{\theta} (t, \theta') \qquad \langle \Sigma, pc, t\rangle \Downarrow^{\theta'} \langle \Sigma', pc', v\rangle}{\langle \Sigma, pc, e\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle}$$

(a) Forcing semantics: $\langle \Sigma, pc, e\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$.

(Thunk)
$$t \Downarrow^{\theta} (t, \theta)$$

(Fun)
$$\lambda x.e \Downarrow^{\theta} (x.e, \theta)$$

(Var)
$$x \Downarrow^{\theta} \theta(x)$$

(App)
$$\frac{e_1 \Downarrow^{\theta} (x.e, \theta') \qquad e_2 \Downarrow^{\theta} v_2 \qquad e \Downarrow^{\theta'[x \mapsto v_2]} v}{e_1\ e_2 \Downarrow^{\theta} v}$$

(b) Pure semantics: $e \Downarrow^{\theta}$ v (selected rules).

(Return)
$$\frac{e \Downarrow^{\theta} v}{\langle \Sigma, pc, \mathbf{return}(e)\rangle \Downarrow^{\theta} \langle \Sigma, pc, v\rangle}$$

(Bind)
$$\frac{\langle \Sigma, pc, e_1\rangle \Downarrow^{\theta} \langle \Sigma', pc', v_1\rangle \qquad \langle \Sigma', pc', e_2\rangle \Downarrow^{\theta[x \mapsto v_1]} \langle \Sigma'', pc'', v\rangle}{\langle \Sigma, pc, \mathbf{bind}(e_1, x.e_2)\rangle \Downarrow^{\theta} \langle \Sigma'', pc'', v\rangle}$$

(ToLabeled)
$$\frac{\langle \Sigma, pc, e\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle}{\langle \Sigma, pc, \mathbf{toLabeled}(e)\rangle \Downarrow^{\theta} \langle \Sigma', pc, \mathbf{Labeled}\ pc'\ v\rangle}$$

(Unlabel)
$$\frac{e \Downarrow^{\theta} \mathbf{Labeled}\ \ell\ v}{\langle \Sigma, pc, \mathbf{unlabel}(e)\rangle \Downarrow^{\theta} \langle \Sigma', pc \sqcup \ell, v\rangle}$$

(LabelOf)
$$\frac{e \Downarrow^{\theta} \mathbf{Labeled}\ \ell\ v}{\langle \Sigma, pc, \mathbf{labelOf}(e)\rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, \ell\rangle}$$

(GetLabel)
$$\langle \Sigma, pc, \mathbf{getLabel}\rangle \Downarrow^{\theta} \langle \Sigma, pc, pc\rangle$$

(Taint)
$$\frac{e \Downarrow^{\theta} \ell}{\langle \Sigma, pc, \mathbf{taint}(e)\rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, ()\rangle}$$

(c) Thunk semantics: $\langle \Sigma, pc, t\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$.

Fig. 7: Big-step semantics for λ^{dCG}.

$$\text{(New)}\quad \frac{e \Downarrow^{\theta} \mathbf{Labeled}\ \ell\ v \qquad pc \sqsubseteq \ell \qquad n = |\Sigma(\ell)|}{\langle \Sigma, pc, \mathbf{new}(e)\rangle \Downarrow^{\theta} \langle \Sigma[\ell \mapsto \Sigma(\ell)[n \mapsto v]], pc, n_\ell\rangle}$$

$$\text{(Read)}\quad \frac{e \Downarrow^{\theta} n_\ell \qquad \Sigma(\ell)[n] = v}{\langle \Sigma, pc, !e\rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, v\rangle}$$

$$\text{(Write)}\quad \frac{e_1 \Downarrow^{\theta} n_{\ell_1} \qquad e_2 \Downarrow^{\theta} \mathbf{Labeled}\ \ell_2\ v \qquad \ell_2 \sqsubseteq \ell_1 \qquad pc \sqsubseteq \ell_1}{\langle \Sigma, pc, e_1 := e_2\rangle \Downarrow^{\theta} \langle \Sigma[\ell_1 \mapsto \Sigma(\ell_1)[n \mapsto v]], pc, ()\rangle}$$

$$\text{(LabelOfRef)}\quad \frac{e \Downarrow^{\theta} n_\ell}{\langle \Sigma, pc, \mathbf{labelOfRef}(e)\rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, \ell\rangle}$$

Fig. 8: Big-step semantics for λ^{dCG} (references).

annotations. They are also *pure*—they do not have access to the store, thus only the security monitor needs to protect against *implicit flows*.

If the pure evaluation reaches a side-effectful computation, i.e., thunk t, it *suspends* the computation by creating a thunk closure that captures the current environment θ (see rule [Thunk]). Notice that *thunk closures* and *function closures* are distinct values created by different rules, [Thunk] and [Fun] respectively.[10] Function application succeeds only when the function evaluates to a function closure (rule [App]). In the thunk semantics, rule [Return] evaluates a pure value embedded in the monad via $\mathbf{return}(\cdot)$ and leaves the state unchanged, while rule [Bind] executes the first computation with the forcing semantics, binds the result in the environment i.e., $\theta[x \mapsto v_1]$, passes it on to the second computation together with the updated state and returns the final result and state. Rule [Unlabel] is interesting. Following the *floating-label* principle, it returns the value wrapped inside the labeled value, i.e., v, and raises the program counter with its label, i.e., $pc \sqcup \ell$, to reflect the fact that new data at security level ℓ is now in scope.

Floating-label based coarse-grained IFC systems like **LIO** suffer from the *label creep* problem, which occurs when the program counter gets over-tainted, e.g., because too many secrets have unlabeled, to the point that no useful further computation can be performed. Primitive $\mathbf{toLabeled}(\cdot)$ provides a mechanism to address this problem by (i) creating a separate context where some sensitive computation can take place and (ii) restoring the original program counter label afterwards. Rule [ToLabeled] formalizes this idea. Notice that the result of

[10] It would have also been possible to define thunk values in terms of function closures using explicit suspension and an opaque wrapper, e.g., **LIO** $(\_.t, \theta)$.

the nested sensitive computation, i.e., v, cannot be simply returned to the lower context—that would be a leak, so $\mathbf{toLabeled}(\cdot)$ wraps that piece of information in a labeled value protected by the final program counter of the sensitive computation, i.e., $\mathbf{Labeled}\ pc'\ v$.[11] Furthermore, notice that pc', the label that tags the result v, is as sensitive as the result itself because the final program counter depends on all the $\mathbf{unlabel}(\cdot)$ operations performed to compute the result. This motivates why primitive $\mathbf{labelOf}(\cdot)$ does not simply project the label from a labeled value, but additionally taints the program counter with the label itself in rule [LabelOf]–a label in a labeled value has sensitivity equal to the label itself, thus the program counter label rises to accommodate reading new sensitive data.

Lastly, rule [GetLabel] returns the value of the program counter, which does not rise (because $pc \sqcup pc = pc$), and rule [Taint] simply taints the program counter with the given label and returns unit (this primitive matches the functionality of $\mathbf{taint}(\cdot)$ in λ^{dFG}). Note that, in λ^{dCG}, $\mathbf{taint}(\cdot)$ takes *only* the label with which the program counter must be tainted whereas, in λ^{dFG}, it additionally requires the expression that must be evaluated in the tainted environment. This difference highlights the *flow-sensitive* nature of the program counter label in λ^{dCG}.

References. λ^{dCG} features *flow-insensitive* labeled references similar to λ^{dFG} and allows programs to create, read, update and inspect the label inside the **LIO** monad (see Figure 8). The API of these primitives takes explicitly labeled values as arguments, by making explicit at the type level, the tagging that occurs in memory, which was left implicit in previous work [144]. Rule [New] creates a reference labeled with the same label annotation as that of the labeled value it receives as an argument, and checks that $pc \sqsubseteq \ell$ in order to avoid implicit flows. Rule [Read] retrieves the content of the reference from the ℓ-labeled memory and returns it. Since this brings data at security level ℓ in scope, the program counter is tainted accordingly, i.e., $pc \sqcup \ell$. Rule [Write] performs security checks analogous to those in λ^{dFG} and updates the content of a given reference and rule [LabelOfRef] returns the label on a reference and taints the context accordingly.

We conclude this section by noting that the forcing and the thunk semantics of λ^{dCG} satisfy Property 2 (*"the final value of the program counter is at least as sensitive as the initial value"*).

Property 2

- *If* $\langle \Sigma, pc, e\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$ *then* $pc \sqsubseteq pc'$.
- *If* $\langle \Sigma, pc, t\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$ *then* $pc \sqsubseteq pc'$.

Proof. By mutual induction on the given evaluation derivations.

[11] [144] have proposed an alternative flow-insensitive primitive, i.e., $\mathbf{toLabeled}(\ell, e)$, which labels the result with the user-assigned label ℓ. The semantics of λ^{dFG} forced us to use $\mathbf{toLabeled}(e)$.

$$\frac{\ell \sqsubseteq L \qquad v_1 \approx_L v_2}{\textbf{Labeled}\ \ell\ v_1 \approx_L \textbf{Labeled}\ \ell\ v_2}\ (\textsc{Labeled}_L) \qquad \frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{\textbf{Labeled}\ \ell_1\ v_1 \approx_L \textbf{Labeled}\ \ell_2\ v_2}\ (\textsc{Labeled}_H)$$

$$\frac{e_1 \equiv_\alpha e_2 \qquad \theta_1 \approx_L \theta_2}{(e_1, \theta_1) \approx_L (e_2, \theta_2)}\ (\textsc{Closure}) \qquad \frac{t_1 \equiv_\alpha t_2 \qquad \theta_1 \approx_L \theta_2}{(t_1, \theta_1) \approx_L (t_2, \theta_2)}\ (\textsc{Thunk}) \qquad \frac{\ell \sqsubseteq L}{n^\ell \approx_L n^\ell}\ (\textsc{Ref}_L)$$

$$\frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{{n_1}^{\ell_1} \approx_L {n_2}^{\ell_2}}\ (\textsc{Ref}_H) \qquad \frac{\Sigma_1 \approx_L \Sigma_2 \qquad pc_1 \not\sqsubseteq L \qquad pc_2 \not\sqsubseteq L}{\langle \Sigma_1, pc_1, v_1 \rangle \approx_L \langle \Sigma_2, pc_2, v_2 \rangle}\ (\textsc{Pc}_H)$$

$$\frac{\Sigma_1 \approx_L \Sigma_2 \qquad pc \sqsubseteq L \qquad v_1 \approx_L v_2}{\langle \Sigma_1, pc, v_1 \rangle \approx_L \langle \Sigma_2, pc, v_2 \rangle}\ (\textsc{Pc}_L)$$

Fig. 9: L-equivalence for λ^{dCG} values (selected rules) and configurations.

3.2 Security

We now prove that λ^{dCG} is secure, i.e., it satisfies *termination-insensitive noninterference*. The meaning of the security condition is intuitively similar to that presented in Section 2.2 for λ^{dFG}— when secret inputs are changed, terminating programs do not produce any publicly observable effect—and based on a similar indistinguishability relation. Figure 9 presents the definition of L-equivalence for the interesting cases only. Firstly, L-equivalence for λ^{dCG} labeled values relates public and secret values analogously to λ^{dFG} values. Specifically, rule [$\textsc{Labeled}_L$] relates public labeled values that share the *same* observable label ($\ell \sqsubseteq L$) and contain related values, i.e., $v_1 \approx_L v_2$, while rule [$\textsc{Labeled}_H$] relates secret labeled values, with arbitrary sensitivity labels not below L ($\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$) and contents. Secondly, L-equivalence relates standard (unlabeled) values homomorphically. For example, values of the sum type are related only as follows: $\textbf{inl}(v_1) \approx_L \textbf{inl}(v_1')$ iff $v_1 \approx_L v_1'$ and $\textbf{inr}(v_2) \approx_L \textbf{inr}(v_2')$ iff $v_2 \approx_L v_2'$. Closures and thunks are related if the function and the monadic computations are α-equivalent and their environments are related, i.e., $\theta_1 \approx_L \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x.\theta_1(x) \approx_L \theta_2(x)$. Labeled references, memories and stores are related by L-equivalence analogously to λ^{dFG}. Lastly, L-equivalence relates *initial* configurations with related stores, equal program counters and α-equivalent expressions (resp. thunks), i.e., $c_1 \approx_L c_2$ iff $c_1 = \langle \Sigma_1, pc_1, e_1 \rangle$, $c_2 = \langle \Sigma_2, pc_2, e_2 \rangle$, $\Sigma_1 \approx_L \Sigma_2$, $pc_1 \equiv pc_2$, and $e_1 \equiv_\alpha e_2$ (resp. $t_1 \equiv_\alpha t_2$ for thunks t_1 and t_2), and *final* configurations with related stores and (i) equal public program counter, i.e., $pc \sqsubseteq L$, and related values [$\textsc{Pc}_L$], or (ii) arbitrary secret public counters, i.e., $pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$, and arbitrary values [$\textsc{Pc}_H$].

We now formally state and prove that λ^{dCG} semantics preserves L-equivalence under L-equivalent environments, i.e., *termination-insensitive non-interference* (TINI).

Theorem 2 (λ^{dCG}-TINI)
If $c_1 \Downarrow^{\theta_1} c_1', c_2 \Downarrow^{\theta_2} c_2', \theta_1 \approx_L \theta_2$ *and* $c_1 \approx_L c_2$ *then* $c_1' \approx_L c_2'$.

Proof. By induction on the derivations.

At this point, we have formalized two calculi—λ^{dFG} and λ^{dCG}—that perform dynamic IFC at *fine* and *coarse* granularity, respectively. While they have some similarities, i.e., they are both functional languages that feature labeled annotated data, references and label introspection primitives, and ensure a termination-insensitive security condition, they also have striking differences. First and foremost, they differ in the number of label annotations—pervasive in λ^{dFG} and optional in λ^{dCG}—with significant implications for the programming model and usability. Second, they differ in the nature of the program counter, *flow-insensitive* in λ^{dFG} and *flow-sensitive* in λ^{dCG}. Third, they differ in the way they deal with side-effects—λ^{dCG} allows side-effectful computations exclusively inside the monad, while λ^{dFG} is *impure*, i.e., any λ^{dFG} expression can modify the state. This difference affects the effort required to implement a system that performs language-based fine- and coarse-grained dynamic IFC. In fact, several coarse-grained IFC languages [30, 63, 123, 124, 131, 149] have been implemented as an embedded domain specific language (EDSL) in a Haskell library with little effort, exploiting the strict control that the host language provides on side-effects. Adapting an existing language to perform fine-grained IFC requires major engineering effort, because several components (all the way from the parser to the runtime system) must be adapted to be label-aware.

In the next two sections we show that—despite their differences—these two calculi are, in fact, equally expressive.

4 Fine- to Coarse-Grained Program Translation

This section presents a provably semantics-preserving program translation from the fine-grained dynamic IFC calculus λ^{dFG} to the coarse-grained calculus λ^{dCG}. At a high level, the translation performs two tasks (i) it embeds the *intrinsic* label annotation of λ^{dFG} values into an *explicitly* labeled λ^{dCG} value via the **Labeled** type constructor and (ii) it restructures λ^{dFG} *side-effectful* expressions into *monadic operations* inside the **LIO** monad. Our type-driven approach starts by formalizing this intuition in the function $\langle\!\langle \cdot \rangle\!\rangle$, which maps the λ^{dFG} type τ to the corresponding λ^{dCG} type $\langle\!\langle \tau \rangle\!\rangle$ (see Figure 10a). The function is defined by induction on types and recursively adds the **Labeled** type constructor to each existing λ^{dFG} type constructor. For the function type $\tau_1 \to \tau_2$, the result is additionally monadic, i.e., $\langle\!\langle \tau_1 \rangle\!\rangle \to \mathbf{LIO} \langle\!\langle \tau_2 \rangle\!\rangle$. This is because the function's body in λ^{dFG} may have side-effects. The translation

$$\langle\!\langle \mathbf{unit} \rangle\!\rangle = \mathbf{Labeled\ unit}$$
$$\langle\!\langle \mathscr{L} \rangle\!\rangle = \mathbf{Labeled}\ \mathscr{L}$$
$$\langle\!\langle \tau_1 \times \tau_2 \rangle\!\rangle = \mathbf{Labeled}\ (\langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle)$$
$$\langle\!\langle \tau_1 + \tau_2 \rangle\!\rangle = \mathbf{Labeled}\ (\langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle)$$
$$\langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle = \mathbf{Labeled}\ (\langle\!\langle \tau_1 \rangle\!\rangle \to \mathbf{LIO}\langle\!\langle \tau_2 \rangle\!\rangle)$$
$$\langle\!\langle \mathbf{Ref}\ \tau \rangle\!\rangle = \mathbf{Labeled}\ (\mathbf{Ref}\langle\!\langle \tau \rangle\!\rangle)$$

(a) Types.

$$\langle\!\langle r^{\ell} \rangle\!\rangle = \mathbf{Labeled}\ \ell\langle\!\langle r \rangle\!\rangle$$
$$\langle\!\langle () \rangle\!\rangle = ()$$
$$\langle\!\langle \ell \rangle\!\rangle = \ell$$
$$\langle\!\langle (v_1, v_2) \rangle\!\rangle = (\langle\!\langle v_1 \rangle\!\rangle, \langle\!\langle v_2 \rangle\!\rangle)$$
$$\langle\!\langle \mathbf{inl}(v) \rangle\!\rangle = \mathbf{inl}(\langle\!\langle v \rangle\!\rangle)$$
$$\langle\!\langle \mathbf{inr}(v) \rangle\!\rangle = \mathbf{inr}(\langle\!\langle v \rangle\!\rangle)$$
$$\langle\!\langle (x.e, \theta) \rangle\!\rangle = (x.\langle\!\langle e \rangle\!\rangle, \langle\!\langle \theta \rangle\!\rangle)$$
$$\langle\!\langle n_{\ell} \rangle\!\rangle = n_{\ell}$$

(b) Values.

Fig. 10: Translation from λ^{dFG} to λ^{dCG}.

for values (Figure 10b) is straightforward. Each λ^{dFG} label tag becomes the label annotation in a λ^{dCG} labeled value. The translation is homomorphic in the constructors on raw values. The translation converts a λ^{dFG} function closure into a λ^{dCG} thunk closure by translating the body of the function to a thunk, i.e., $\langle\!\langle e \rangle\!\rangle$ (see below), and translating the environment pointwise, i.e., $\langle\!\langle \theta \rangle\!\rangle = \lambda x.\langle\!\langle \theta(x) \rangle\!\rangle$.

Expressions. We show the translation of λ^{dFG} expressions to λ^{dCG} monadic thunks in Figure 11. We use the standard **do** notation for readability.[12] First, notice that the translation of all constructs occurs inside a $\mathbf{toLabeled}(\cdot)$ block. This achieves two goals, (i) it ensures that the value that results from a translated expression is *explicitly* labeled and (ii) it creates an isolated nested context where the translated thunk can execute without raising the program counter label at the top level. Inside the $\mathbf{toLabeled}(\cdot)$ block, the program counter label may rise, e.g., when some intermediate result is unlabeled, and the translation relies on **LIO**'s floating-label mechanism to track dependencies between data of different security levels. In particular, we will show later that the value of the program counter label at the end of each nested block coincides with the label annotation of the λ^{dFG} value that the original expression evaluates to. For example, introduction forms of ground values (unit, labels, and functions) are simply returned inside the $\mathbf{toLabeled}(\cdot)$ block so that they get tagged with the current value of the program counter label just as in the corresponding λ^{dFG} introduction rules ([Label,Unit,Fun]). Introduction forms of compounds values such as $\mathbf{inl}(e)$, $\mathbf{inr}(e)$ and (e_1, e_2) follow the same principle. The translation simply nests the translations of the nested expressions inside the same constructor, without raising the program counter label. This matches the behavior of the corresponding λ^{dFG} rules [Inl,Inr,Pair].[13] For ex-

[12] Syntax $\mathbf{do}\ \ x \leftarrow e_1; e_2$ desugars to $\mathbf{bind}(e_1, x.e_2)$ and syntax $e_1; e_2$ to $\mathbf{bind}(e_1, \_.e_2)$.

[13] We name a variable lv if it gets bound to a labeled value, i.e., to indicate that the variable has type $\mathbf{Labeled}\ \tau$.

$$\langle\!\langle () \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(()))$$

$$\langle\!\langle \ell \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(\ell))$$

$$\langle\!\langle (\lambda x.e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ \mathbf{return}(\lambda x.\langle\!\langle e \rangle\!\rangle))$$

$$\langle\!\langle \mathbf{inl}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ le \leftarrow \langle\!\langle e \rangle\!\rangle\ \mathbf{return}(\mathbf{inl}(lv)))$$

$$\langle\!\langle \mathbf{inr}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ le \leftarrow \langle\!\langle e \rangle\!\rangle\ \mathbf{return}(\mathbf{inr}(lv)))$$

$$\langle\!\langle (e_1, e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\ lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\ \mathbf{return}(lv_1, lv_2))$$

$$\langle\!\langle x \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{unlabel}(x))$$

$$\langle\!\langle e_1\ e_2 \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\ lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\ v_1 \leftarrow \mathbf{unlabel}(lv_1)\ lv \leftarrow v_1\ lv_2\ \mathbf{unlabel}(lv))$$

$$\langle\!\langle \mathbf{case}(e, x.e_1, x.e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv \leftarrow \langle\!\langle e \rangle\!\rangle\ v \leftarrow \mathbf{unlabel}(lv)\ lv' \leftarrow \mathbf{case}(v, x.\langle\!\langle e_1 \rangle\!\rangle, x.\langle\!\langle e_2 \rangle\!\rangle)\ \mathbf{unlabel}(lv'))$$

$$\langle\!\langle \mathbf{fst}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv \leftarrow \langle\!\langle e \rangle\!\rangle\ v \leftarrow \mathbf{unlabel}(lv)\ \mathbf{unlabel}(\mathbf{fst}(v)))$$

$$\langle\!\langle \mathbf{snd}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv \leftarrow \langle\!\langle e \rangle\!\rangle\ v \leftarrow \mathbf{unlabel}(lv)\ \mathbf{unlabel}(\mathbf{snd}(v)))$$

$$\langle\!\langle \mathbf{taint}(e_1, e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\ v_1 \leftarrow \mathbf{unlabel}(lv_1)\ \mathbf{taint}(v_1)\ lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\ \mathbf{unlabel}(lv_2))$$

$$\langle\!\langle \mathbf{labelOf}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\ lv \leftarrow \langle\!\langle e \rangle\!\rangle\ \mathbf{labelOf}(lv))$$

$$\langle\!\langle \mathbf{getLabel} \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{getLabel})$$

Fig. 11: Translation from λ^{dFG} to λ^{dCG} (expressions).

ample, the λ^{dFG} reduction $((), ()) \Downarrow_L^{\varnothing} ((){}^{L}, (){}^{L}){}^{L}$ maps to a λ^{dCG} reduction that yields $\mathbf{Labeled}\ L\ (\mathbf{Labeled}\ L\ (), \mathbf{Labeled}\ L\ ())$ when started with program counter label L.

The translation of variables gives some insight into how the λ^{dCG} floating-label mechanism can simulate λ^{dFG}'s tainting approach. First, the type-driven approach set out in Figure 10a demands that functions take only labeled values as arguments, so the variables in the source program are always associated to a labeled value in the translated program. The values that correspond to these variables are stored in the environment θ and translated separately, e.g., if $\theta(x) = r^{\ell}$ in λ^{dFG}, then x gets bound to $\langle\!\langle r^{\ell} \rangle\!\rangle = \mathbf{Labeled}\ \ell\langle\!\langle r \rangle\!\rangle$ when translated to λ^{dCG}. Thus, the translation converts a variable, say x, to $\mathbf{toLabeled}(\mathbf{unlabel}(x))$, so that its label gets tainted with the current program counter label. More precisely, $\mathbf{unlabel}(x)$ retrieves the labeled value associated with the variable, i.e., $\mathbf{Labeled}\ \ell\langle\!\langle r \rangle\!\rangle$, taints the program counter with its label to make it $pc \sqcup \ell$, and returns the content, i.e., $\langle\!\langle r \rangle\!\rangle$. Since $\mathbf{unlabel}(x)$ occurs inside a $\mathbf{toLabeled}(\cdot)$ block, the code above results in

$$\frac{\Gamma \setminus \overline{x} \vdash e : \tau}{\Gamma \vdash \mathit{wken}\ \overline{x}\ e : \tau}\ (\textsc{WkenType}) \qquad \frac{e \Downarrow^{\theta \setminus \overline{x}} v}{\mathit{wken}\ \overline{x}\ e \Downarrow^{\theta} v}\ (\textsc{Wken})$$

Fig. 12: Typing and semantics rules of wken for λ^{dCG}.

$\mathbf{Labeled}\ (pc \sqcup \ell)\langle\!\langle r \rangle\!\rangle$ when evaluated, matching precisely the tainting behavior of λ^{dFG} rule [Var], i.e., $x \Downarrow_{pc}^{\theta[x \mapsto r^{\ell}]} r^{pc \sqcup \ell}$.

The elimination forms for other types (function application, pair projections and case analysis) follow the same approach. For example, the code that translates a function application $e_1\ e_2$ first executes the code that computes the translated function, i.e., $lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle$, then the code that computes the argument, i.e., $lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle$ and then retrieves the function from the first labeled value, i.e., $v_1 \leftarrow \mathbf{unlabel}(lv_1)$.[14] The function v_1 applied to the labeled argument lv_2 gives a computation that gets executed and returns a labeled value lv that gets unlabeled to expose the final result (the surrounding $\mathbf{toLabeled}(\cdot)$ wraps it again right away). The translation of case analysis is analogous. The translation of pair projections first converts the λ^{dFG} pair into a computation that gives a λ^{dCG} labeled pair of labeled values, say $\mathbf{Labeled}\ \ell\ (\mathbf{Labeled}\ \ell_1\langle\!\langle r_1 \rangle\!\rangle, \mathbf{Labeled}\ \ell_2\langle\!\langle r_2 \rangle\!\rangle)$ and removes the label tag on the pair via $\mathbf{unlabel}$, thus raising the program counter label to $pc \sqcup \ell$. Then, it projects the appropriate component and unlabels it, thus tainting the program counter label even further with the label of either the first or the second component. This coincides with the tainting mechanism of λ^{dFG} for projection rules, e.g., in rule [Fst] where $\mathbf{fst}(e) \Downarrow_{pc}^{\theta} {r_1}^{pc \sqcup \ell \sqcup \ell_1}$ if $e \Downarrow_{pc}^{\theta} ({r_1}^{\ell_1}, {r_2}^{\ell_2})^{\ell}$.

Lastly, translating $\mathbf{taint}(e_1, e_2)$ requires (i) translating the expression e_1 that gives the label, (ii) using $\mathbf{taint}(\cdot)$ from λ^{dCG} to explicitly taint the program counter label with the label that e_1 gives, and (iii) translating the second argument e_2 to execute in the tainted context and unlabeling the result. The construct $\mathbf{labelOf}(e)$ of λ^{dFG} uses the corresponding λ^{dCG} primitive applied on the corresponding labeled value, say $\mathbf{Labeled}\ \ell\langle\!\langle r \rangle\!\rangle$, obtained from the translated expression. Notice that $\mathbf{labelOf}(\cdot)$ taints the program counter label in λ^{dCG}, which rises to $pc \sqcup \ell$, so the code just described results in $\mathbf{Labeled}\ (pc \sqcup \ell)\ \ell$, which corresponds to the translation of the result in λ^{dFG}, i.e., $\langle\!\langle \ell^{\ell} \rangle\!\rangle = \mathbf{Labeled}\ \ell\ \ell$ because $pc \sqcup \ell \equiv \ell$, since $pc \sqsubseteq \ell$ from Property 1. The translation of $\mathbf{getLabel}$ follows naturally by simply wrap-

[14] Notice that it is incorrect to unlabel the function before translating the argument, because that would taint the program counter label, which would raise at level, say $pc \sqcup \ell$, and affect the code that translates the argument, which was to be evaluated with program counter label equal to pc by the original *flow-insensitive* λ^{dFG} rule [App] for function application.

$$
\begin{array}{l}
\langle\!\langle \mathbf{new}(e) \rangle\!\rangle = \\
\quad \mathbf{toLabeled}(\ \mathbf{do} \\
\qquad lv \leftarrow \langle\!\langle e \rangle\!\rangle \\
\qquad \mathbf{new}(lv)) \\
\langle\!\langle\ !\, e \rangle\!\rangle = \\
\quad \mathbf{toLabeled}(\ \mathbf{do} \\
\qquad lr \leftarrow \langle\!\langle e \rangle\!\rangle \\
\qquad r \leftarrow \mathbf{unlabel}(lv) \\
\qquad !\, r)
\end{array}
\qquad
\begin{array}{l}
\langle\!\langle e_1 := e_2 \rangle\!\rangle = \\
\quad \mathbf{toLabeled}(\ \mathbf{do} \\
\qquad lr \leftarrow \langle\!\langle e_1 \rangle\!\rangle \\
\qquad lv \leftarrow \langle\!\langle e_2 \rangle\!\rangle \\
\qquad r \leftarrow \mathbf{unlabel}(lr) \\
\qquad r := lv) \\
\quad \mathbf{toLabeled}(\mathbf{return}())
\end{array}
\qquad
\begin{array}{l}
\langle\!\langle \mathbf{labelOfRef}(e) \rangle\!\rangle = \\
\quad \mathbf{toLabeled}(\ \mathbf{do} \\
\qquad lr \leftarrow \langle\!\langle e \rangle\!\rangle \\
\qquad r \leftarrow \mathbf{unlabel}(lv) \\
\qquad \mathbf{labelOfRef}(r))
\end{array}
$$

Fig. 13: Translation λ^{dFG} to λ^{dCG} (references).

ping λ^{dCG}'s $\mathbf{getLabel}$ inside a $\mathbf{toLabeled}(\cdot)$, which correctly returns the program counter label labeled with itself, i.e., $\mathbf{Labeled}\ pc\ pc$.

Note on Environments. The semantics rules of λ^{dFG} and λ^{dCG} feature an environment θ for input values that gets extended with intermediate values during program evaluation and that may be captured inside a closure. Unfortunately, this capturing behavior is undesirable for our program translation. The program translation defined above introduces temporary auxiliary variables that carry the value of intermediate results, e.g., the labeled value obtained from running a computation that translates some λ^{dFG} expression. When the translated program is executed, these values end up in the environment, e.g., by means of rules [App] and [Bind], and mix with the input values of the source program and output values as well, thus complicating the correctness statement of the translation, which now has to account for those extra variables as well. In order to avoid this nuisance, we employ a special form of weakening that allows shrinking the environment at run-time and removing spurious values that are not needed in the rest of the program. In particular, expression $wken\ \overline{x}\ e$ has the same type as e if variables $\overline{x}$ are not free in e, see the formal typing rule [WkenType] in Figure 12. At run-time, the expression $wken\ \overline{x}\ e$ evaluates e in an environment from which variables $\overline{x}$ have been dropped, so that they do not get captured in any closure created during the execution of e. Rule [Wken] is part of the pure semantics of λ^{dCG}—the semantics of λ^{dFG} includes an analogous rule (the issue of contaminated environments arises in the translations in both directions, thus both calculi feature $wken$). We remark that this expedient is not essential—we can avoid it by slightly complicating the correctness statement to explicitly account for those extra variables. Nor is this expedient particularly interesting. In fact, we omit $wken$ from the code of the program translations to avoid clutter (our mechanization includes $wken$ in the appropriate places).

References. Figure 13 shows the program translation of λ^{dFG} primitives that access the store via references. The translation of λ^{dFG} values wraps references in λ^{dCG} labeled values (Figure 10b), so the translations of Figure 13 take care of boxing and unboxing references. The translation of $\mathbf{new}(e)$ has a top-level

$\mathbf{toLabeled}(\cdot)$ block that simply translates the content ($lv \leftarrow \langle\!\langle e \rangle\!\rangle$) and puts it in a new reference ($\mathbf{new}(lv)$). The λ^{dCG} rule [New] (Figure 8) assigns the label of the translated content to the new reference, which also gets labeled with the original program counter label[15], just as in the λ^{dFG} rule [New] (Figure 4). In λ^{dFG}, rule [Read] reads from a reference ${n_\ell}^{\ell'}$ at security level ℓ' that points to the ℓ-labeled memory, and returns the content $\Sigma(\ell)[n]^{\ell \sqcup \ell'}$ at level $\ell \sqcup \ell'$. Similarly, the translation creates a $\mathbf{toLabeled}(\cdot)$ block that executes to get a labeled reference $lr = \mathbf{Labeled}\ \ell'\ n_\ell$, extracts the reference n_ℓ ($r \leftarrow \mathbf{unlabel}(lr)$) tainting the program counter label with ℓ', and then reads the reference's content further tainting the program counter label with ℓ as well. The code that translates and updates a reference consists of two $\mathbf{toLabeled}(\cdot)$ blocks. The first block is responsible for the update: it extracts the labeled reference and the labeled new content (lr and lv resp.), extracts the reference from the labeled value ($r \leftarrow \mathbf{unlabel}(lr)$) and updates it ($r := lv$). The second block, $\mathbf{toLabeled}(\mathbf{return}())$, returns unit at security level pc, i.e., $\mathbf{Labeled}\ pc\ ()$, similar to the λ^{dFG} rule [Write]. The translation of $\mathbf{labelOfRef}(e)$ extracts the reference and projects its label via the λ^{dCG} primitive $\mathbf{labelOfRef}(\cdot)$, which additionally taints the program counter with the label itself, similar to the λ^{dFG} rule [LabelOfRef].

4.1 Correctness

In this section, we establish some desirable properties of the λ^{dFG}-to-λ^{dCG} translation defined above. These properties include type and semantics preservation as well as recovery of non-interference—a meta criterion that rules out a class of semantically correct (semantics preserving), yet elusive translations that do not preserve the meaning of security labels [16, 121].

We start by showing that the program translation preserves typing. The type translation for typing contexts Γ is pointwise, i.e., $\langle\!\langle \Gamma \rangle\!\rangle = \lambda x.\langle\!\langle \Gamma(x) \rangle\!\rangle$.

Lemma 1 (Type Preservation).
Given a well-typed λ^{dFG} expression, i.e., $\Gamma \vdash e : \tau$, the translated λ^{dCG} expression is also well-typed, i.e., $\langle\!\langle \Gamma \rangle\!\rangle \vdash \langle\!\langle e \rangle\!\rangle : \mathbf{LIO}\langle\!\langle \tau \rangle\!\rangle$.

Proof 1 *By induction on the given typing derivation.*

The main correctness criterion for the translation is semantics preservation. Intuitively, proving this theorem ensures that the program translation preserves the meaning of secure λ^{dFG} programs when translated and executed with λ^{dCG} semantics (under a translated environment). In the theorem below[16], the translation of stores and memories is pointwise, i.e., $\langle\!\langle \Sigma \rangle\!\rangle = \lambda \ell.\langle\!\langle \Sigma(\ell) \rangle\!\rangle$, and $\langle\!\langle [\,] \rangle\!\rangle = [\,]$ and $\langle\!\langle r : M \rangle\!\rangle = \langle\!\langle r \rangle\!\rangle : \langle\!\langle M \rangle\!\rangle$ for each ℓ-labeled memory M. Furthermore, notice that in the translation, the initial and final program counter

[15] The nested block does not execute any $\mathbf{unlabel}(\cdot)$ nor $\mathbf{taint}(\cdot)$.

[16] The proof of Theorem 3 requires the (often used) axiom of functional extensionality in our mechanized proofs.

labels are the same. This establishes that the program translation preserves the flow-insensitive program counter label of λ^{dFG} (by means of primitive $\mathbf{toLabeled}(\cdot)$).

Theorem 3 (Semantics Preservation of $\langle\!\langle \cdot \rangle\!\rangle : \lambda^{dFG} \to \lambda^{dCG}$)
If $\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v\rangle$, *then* $\langle \langle\!\langle \Sigma \rangle\!\rangle, pc, \langle\!\langle e \rangle\!\rangle\rangle \Downarrow^{\langle\!\langle \theta \rangle\!\rangle} \langle \langle\!\langle \Sigma' \rangle\!\rangle, pc, \langle\!\langle v \rangle\!\rangle\rangle$.

Proof. By induction on the given evaluation derivation using basic properties of the security lattice and of the translation function.

Recovery of non-interference. We conclude this section by constructing a proof of termination-insensitive non-interference for λ^{dFG} (Theorem 1) from the corresponding theorem for λ^{dCG} (Theorem 2), using the semantics preserving translation (Theorem 3), together with a property that the translation preserves L-equivalence as well (Lemmas 2 and 3). Doing so ensures that the meaning of labels is preserved by the translation [16, 121]. In the absence of such an artifact, one could devise a semantics-preserving translation that simply does not use the security features of the target language. While technically correct (i.e., semantics preserving), the translation would not be meaningful from the perspective of security.[17] The following lemma shows that the translation of λ^{dFG} *initial* configurations, defined as $\langle\!\langle c \rangle\!\rangle^{pc} = \langle \langle\!\langle \Sigma \rangle\!\rangle, pc, \langle\!\langle e \rangle\!\rangle\rangle$ if $c = \langle \Sigma, e\rangle$, preserves L-equivalence by lifting L-equivalence from source to target and back.

Lemma 2. *For all labels* pc, $c_1 \approx_L c_2$ *if and only if* $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2 \rangle\!\rangle^{pc}$.

Proof. By definition of L-equivalence for initial configurations in both directions (Sections 2.2 and 3.2), using injectivity of the translation function, i.e., if $\langle\!\langle e_1 \rangle\!\rangle \equiv_\alpha \langle\!\langle e_2 \rangle\!\rangle$ then $e_1 \equiv_\alpha e_2$, in the *if* direction, and by mutually proving similar lemmas for all categories: for stores, i.e., $\Sigma_1 \approx_L \Sigma_2$ iff $\langle\!\langle \Sigma_1 \rangle\!\rangle \approx_L \langle\!\langle \Sigma_2 \rangle\!\rangle$, for memories, i.e., $M_1 \approx_L M_2$ iff $\langle\!\langle M_1 \rangle\!\rangle \approx_L \langle\!\langle M_2 \rangle\!\rangle$, for environments, i.e., $\theta_1 \approx_L \theta_2$ iff $\langle\!\langle \theta_1 \rangle\!\rangle \approx_L \langle\!\langle \theta_2 \rangle\!\rangle$, for values, i.e., $v_1 \approx_L v_2$ iff $\langle\!\langle v_1 \rangle\!\rangle \approx_L \langle\!\langle v_2 \rangle\!\rangle$, and for raw values, i.e., $r_1 \approx_L r_2$ iff $\langle\!\langle r_1 \rangle\!\rangle \approx_L \langle\!\langle r_2 \rangle\!\rangle$.

The following lemma recovers L-equivalence of *source* final configurations by back-translating L-equivalence of *target* final configurations. We define the translation for λ^{dFG} *final* configurations as $\langle\!\langle c \rangle\!\rangle^{pc} = \langle \langle\!\langle \Sigma \rangle\!\rangle, pc, \langle\!\langle v \rangle\!\rangle\rangle$ if $c = \langle \Sigma, v\rangle$.

Lemma 3. *Let* $c_1 = \langle \Sigma_1, {r_1}^{\ell_1}\rangle$, $c_2 = \langle \Sigma_2, {r_2}^{\ell_2}\rangle$ *be* λ^{dFG} *final configurations. For all program counter label* pc, *such that* $pc \sqsubseteq \ell_1$ *and* $pc \sqsubseteq \ell_2$, *if* $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2 \rangle\!\rangle^{pc}$ *then* $c_1 \approx_L c_2$.

[17] Note that such bogus translations are also ruled out due to the need to preserve the outcome of any label introspection. Nonetheless, building this proof artifact increases our confidence in the robustness of our translation. In contrast, if the enforcement of IFC is *static*, then there is no label introspection, and this proof artifact is extremely important, as argued in prior work [16, 121].

$$
\begin{aligned}
&[\![\mathscr{L}]\!] = \mathscr{L}\\
&[\![\mathbf{unit}]\!] = \mathbf{unit}\\
&[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]\\
&[\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] + [\![\tau_2]\!]\\
&[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]\\
&[\![\mathbf{Ref}\ \tau]\!] = \mathbf{Ref}[\![\tau]\!]\\
&[\![\mathbf{Labeled}\ \tau]\!] = \mathscr{L} \times [\![\tau]\!]\\
&[\![\mathbf{LIO}\ \tau]\!] = \mathbf{unit} \to [\![\tau]\!]
\end{aligned}
$$

(a) Types.

$$
\begin{aligned}
&[\![()]\!]^{pc} = ()^{pc}\\
&[\![\ell]\!]^{pc} = \ell^{pc}\\
&[\![\mathbf{inl}(v)]\!]^{pc} = \mathbf{inl}([\![v]\!]^{pc})^{pc}\\
&[\![\mathbf{inr}(v)]\!]^{pc} = \mathbf{inr}([\![v]\!]^{pc})^{pc}\\
&[\![(v_1, v_2)]\!]^{pc} = ([\![v_1]\!]^{pc}, [\![v_2]\!]^{pc})^{pc}\\
&[\![(x.e, \theta)]\!]^{pc} = (x.[\![e]\!], [\![\theta]\!]^{pc})^{pc}\\
&[\![(t, \theta)]\!]^{pc} = (\_.[\![t]\!], [\![\theta]\!]^{pc})^{pc}\\
&[\![\mathbf{Labeled}\ \ell\ v]\!]^{pc} = (\ell^{\ell}, [\![v]\!]^{\ell})^{pc}\\
&[\![n_\ell]\!]^{pc} = (n_\ell)^{pc}
\end{aligned}
$$

(b) Values.

Fig. 14: Translation from λ^{dCG} to λ^{dFG} (part I).

Proof By case analysis on the L-equivalence relation of the target final configurations, two cases follow. First, we recover L-equivalence of the source stores, i.e., $\Sigma_1 \approx_L \Sigma_2$, from that of the target stores, i.e., $\langle\!\langle \Sigma_1 \rangle\!\rangle \approx_L \langle\!\langle \Sigma_2 \rangle\!\rangle$ from $\langle\!\langle c_1 \rangle\!\rangle \approx_L \langle\!\langle c_2 \rangle\!\rangle$ in both cases. Then, the program counter in the target configurations is either (i) *above* the attacker's level [Pc$_H$], i.e., $pc \not\sqsubseteq L$, and the source values are L-equivalent, i.e., ${r_1}^{\ell_1} \approx_L {r_2}^{\ell_2}$ by rule [Value$_H$] applied to $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$ (from $pc \not\sqsubseteq L$ and, respectively, $pc \sqsubseteq \ell_1$ and $pc \sqsubseteq \ell_2$), or (ii) *below* the attacker's level [Pc$_L$], i.e., $pc \sqsubseteq L$, then $\langle\!\langle {r_1}^{\ell_1} \rangle\!\rangle \approx_L \langle\!\langle {r_2}^{\ell_2} \rangle\!\rangle$ and the source values are L-equivalent, i.e., ${r_1}^{\ell_1} \approx_L {r_2}^{\ell_2}$, by Lemma 2 for values.

Theorem 4 (λ^{dFG}-TINI via $\langle\!\langle \cdot \rangle\!\rangle$)
If $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ *and* $c_1 \approx_L c_2$ *then* $c_1' \approx_L c_2'$.

Proof. We start by applying the fine to coarse grained program translation to the initial configurations and environments. By Theorem 3 (semantics preservation), we derive the corresponding λ^{dCG} reductions, i.e., $\langle\!\langle c_1 \rangle\!\rangle^{pc} \Downarrow^{\langle\!\langle \theta_1 \rangle\!\rangle} \langle\!\langle c_1' \rangle\!\rangle^{pc}$ and $\langle\!\langle c_2 \rangle\!\rangle^{pc} \Downarrow^{\langle\!\langle \theta_2 \rangle\!\rangle} \langle\!\langle c_2' \rangle\!\rangle^{pc}$. Then, we lift L-equivalence of the initial configurations and environments from *source* to *target*, i.e., from $c_1 \approx_L c_2$ to $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2 \rangle\!\rangle^{pc}$ and from $\theta_1 \approx_L \theta_2$ to $\langle\!\langle \theta_1 \rangle\!\rangle \approx_L \langle\!\langle \theta_2 \rangle\!\rangle$ (Lemma 2), and apply λ^{dCG}*-TINI* (Theorem 2) to obtain L-equivalence of the *target* final configurations, i.e., $\langle\!\langle c_1' \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2' \rangle\!\rangle^{pc}$. Finally, we recover L-equivalence of the final configurations from *target* to *source*, i.e., from $\langle\!\langle c_1' \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2' \rangle\!\rangle^{pc}$ to $c_1' \approx_L c_2'$, via Lemma 3, applied to $c_1' = \langle \Sigma_1, {r_1}^{\ell_1} \rangle$ and $c_2' = \langle \Sigma_2, {r_2}^{\ell_2} \rangle$, and where $pc \sqsubseteq \ell_1$ and $pc \sqsubseteq \ell_2$ by Property 1 applied to the source reductions, i.e., $c_1 \Downarrow_{pc}^{\theta_1} c_1'$ and $c_2 \Downarrow_{pc}^{\theta_2} c_2'$.

5 Coarse- to Fine-Grained Program Translation

We now show a verified program translation in the opposite direction—from the coarse grained calculus λ^{dCG} to the fine grained calculus λ^{dFG}. The translation in this direction is more involved—a program in λ^{dFG} contains strictly more information than its counterpart in λ^{dCG}, namely the extra *intrinsic* label

annotations that tag every value. The challenge in constructing this translation is two-fold. On one hand, the translation must come up with labels for all values. However, it is not always possible to do this statically during the translation: Often, the labels depend on input values and arise at run-time with intermediate results since the λ^{dFG} calculus is designed to compute and attach labels at run-time. On the other hand, the translation cannot conservatively under-approximate the values of labels[18]—λ^{dCG} and λ^{dFG} have label introspection so, in order to get semantics preservation, labels must be preserved precisely. Intuitively, we solve this impasse by crafting a program translation that (i) preserves the labels that can be inspected by λ^{dCG} and (ii) lets the λ^{dFG} semantics compute the remaining label annotations automatically—we account for those labels with a *cross-language* relation that represents semantic equivalence between λ^{dFG} and λ^{dCG} modulo extra annotations (Section 5.1). The fact that the source program in λ^{dCG} cannot inspect those labels—they have no value counterpart in the source λ^{dCG} program—facilitates this aspect of the translation. We elaborate more on the technical details later.

At a high level, an interesting aspect of the translation (that informally attests that it is indeed semantics-preserving) is that it encodes the *flow-sensitive* program counter of the source λ^{dCG} program into the label annotation of the λ^{dFG} value that results from executing the translated program. For example, if a λ^{dCG} monadic expression starts with program counter label pc and results in some value, say $\mathbf{true}$, and final program counter pc', then the translated λ^{dFG} expression, starting with the same program counter label pc, computes the *same* value (modulo extra label annotations) at the same security level pc', i.e., the value $\mathbf{true}^{pc'}$. This encoding requires keeping the value of the program counter label in the source program synchronized with the program counter label in the target program, by loosening the fine-grained precision of λ^{dFG} at run-time in a controlled way.

Types. The λ^{dCG}-to-λ^{dFG} translation follows the same type-driven approach used in the other direction, starting from the function $[\![\cdot]\!]$ in Figure 14a, that translates a λ^{dFG} type τ into the corresponding λ^{dCG} type $[\![\tau]\!]$. The translation is defined by induction on τ and preserves all the type constructors standard types. Only the cases corresponding to λ^{dCG}-specific types are interesting. In particular, it converts *explicitly* labeled types, i.e., $\mathbf{Labeled}\ \tau$, to a standard pair type in λ^{dFG}, i.e., $(\mathscr{L} \times [\![\tau]\!])$, where the first component is the label and the second component the content of type τ. Type $\mathbf{LIO}\ \tau$ becomes a *suspension* in λ^{dFG}, i.e., the function type $\mathbf{unit} \to [\![\tau]\!]$ that delays a computation and that can be forced by simply applying it to the unit value $()$.

Values. The translation of values follows the type translation, as shown in Figure 14b. Notice that the translation is indexed by the program counter

[18] In contrast, previous work on *static* type-based fine-to-coarse grained translation safely under-approximates the label annotations in types with $\bot$ [121]. The proof of type preservation of the translation recovers the actual labels via *subtyping*.

$$
\begin{aligned}
&[\![()]\!] = () \\
&[\![\ell]\!] = \ell \\
&[\![x]\!] = x \\
&[\![\lambda x.e]\!] = \lambda x.[\![e]\!] \\
&[\![e_1\ e_2]\!] = [\![e_1]\!][\![e_2]\!] \\
&[\![(e_1, e_2)]\!] = ([\![e_1]\!], [\![e_2]\!]) \\
&[\![\mathbf{fst}(e)]\!] = \mathbf{fst}([\![e]\!]) \\
&[\![\mathbf{snd}(e)]\!] = \mathbf{snd}([\![e]\!]) \\
&[\![\mathbf{inl}(e)]\!] = \mathbf{inl}([\![e]\!]) \\
&[\![\mathbf{inr}(e)]\!] = \mathbf{inr}([\![e]\!]) \\
&[\![\mathbf{case}\ (e, x.e_1, x.e_2)]\!] \\
&\quad = \mathbf{case}\ ([\![e]\!], x.[\![e_1]\!], x.[\![e_2]\!]) \\
&[\![t]\!] = \lambda\_.[\![t]\!]
\end{aligned}
$$

(a) Expressions.

$$
\begin{aligned}
&[\![\mathbf{return}(e)]\!] = [\![e]\!] \\
&[\![\mathbf{bind}(e_1, x.e_2)]\!] = \\
&\quad \mathbf{let}\ x = [\![e_1]\!]()\ \mathbf{in} \\
&\qquad \mathbf{taint}(\mathbf{labelOf}(x), [\![e_2]\!]()) \\
&[\![\mathbf{unlabel}(e)]\!] = \\
&\quad \mathbf{let}\ x = [\![e]\!]\mathbf{in} \\
&\qquad \mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x)) \\
&[\![\mathbf{toLabeled}(e)]\!] = \\
&\quad \mathbf{let}\ x = [\![e]\!]()\ \mathbf{in} \\
&\qquad (\mathbf{labelOf}(x), x) \\
&[\![\mathbf{labelOf}(e)]\!] = \mathbf{fst}([\![e]\!]) \\
&[\![\mathbf{getLabel}]\!] = \mathbf{getLabel} \\
&[\![\mathbf{taint}(e)]\!] = \mathbf{taint}([\![e]\!], ())
\end{aligned}
$$

(b) Thunks.

Fig. 15: Translation from λ^{dCG} to λ^{dFG} (part II).

label (the translation is written $[\![v]\!]^{pc}$), which converts the λ^{dCG} value v in scope of a computation protected by security level pc to the corresponding fully label-annotated λ^{dFG} value. The translation is pretty straightforward and uses the program counter label to tag each value, following the λ^{dCG} principle that the program counter label protects every value in scope that is not explicitly labeled. The translation converts a λ^{dCG} function closure into a corresponding λ^{dFG} function closure by translating the body of the function to a λ^{dFG} expression (see below) and translating the environment pointwise, i.e., $[\![\theta]\!]^{pc} = \lambda x.[\![\theta(x)]\!]^{pc}$. A thunk value or a *thunk closure*, which denotes a suspended side-effecful computation, is also converted into a λ^{dFG} function closure. Technically, the translation would need to introduce a *fresh variable* that would get bound to unit when the suspension gets forced. However, the argument to the suspension does not have any purpose, so we do not bother with giving a name to it and write $\_.[\![t]\!]$ instead. (In our mechanized proofs we employ unnamed De Bruijn indexes and this issue does not arise.) The translation converts an explicitly labeled value $\mathbf{Labeled}\ \ell\ v$, into a labeled pair at security level pc, i.e., $(\ell^{\ell}, [\![v]\!]^{\ell})^{pc}$. The pair consists of the label ℓ tagged with itself, and the value translated at a security level equal to the label annotation, i.e., $[\![v]\!]^{\ell}$. Notice that tagging the label with itself allows us to translate the λ^{dCG} (label introspection) primitive $\mathbf{labelOf}(\cdot)$ by simply projecting the first component, thus preserving the label and its security level across the translation.

Expressions and Thunks. The translation of pure expressions (Figure 15a) is trivial: it is homomorphic in all constructs, mirroring the type translation. The translation of a thunk expression t builds a suspension explicitly with a λ-abstraction (the name of the variable is again irrelevant, thus we omit it as explained above), and carries on by translating the thunk itself according to the

$$
\begin{array}{ll}
[\![\mathbf{new}(e)]\!] = & [\![e_1 := e_2]\!] = [\![e_1]\!] := \mathbf{snd}([\![e_2]\!]) \\
\quad \mathbf{let}\ x = [\![e]\!] \mathbf{in} & [\![\,!\,e]\!] = ![\![e]\!] \\
\quad\quad \mathbf{new}(\mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x))) & [\![\mathbf{labelOfRef}(e)]\!] = \mathbf{labelOfRef}([\![e]\!])
\end{array}
$$

Fig. 16: Translation from λ^{dCG} to λ^{dFG} (references).

definition in Figure 15b. The thunk $\mathbf{return}(e)$ becomes $[\![e]\!]$, since $\mathbf{return}(\cdot)$ does not have any side-effect. When two monadic computations are combined via $\mathbf{bind}(e_1, x.e_2)$, the translation (i) converts the first computation to a suspension and forces it by applying unit ($[\![e_1]\!]()$), (ii) binds the result to x and passes it to the second computation[19], which is also converted, forced, and, *importantly*, iii) executed with a program counter label tainted with the security level of the result of the first computation ($\mathbf{taint}(\mathbf{labelOf}(x), [\![e_2]\!]())$). Notice that $\mathbf{taint}(\cdot)$ is essential to ensure that the second computation executes with the program counter label set to the correct value—the precision of the fine-grained system would otherwise retain the initial lower program counter label according to rule [App] and the value of the program counter labels in the source and target programs would differ in the remaining execution.

Similarly, the translation of $\mathbf{unlabel}(e)$ first translates the labeled expression e (the translated expression does not need to be forced because it is not of a monadic type), binds its result to x and then projects the content in a context tainted with its label, as in $\mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x))$. This closely follows λ^{dCG}'s [Unlabel] rule. The translation of $\mathbf{toLabeled}(e)$ forces the nested computation with $[\![e]\!]()$, binds its result to x and creates the pair $(\mathbf{labelOf}(x), x)$, which corresponds to the labeled value obtained in λ^{dCG} via rule [ToLabeled]. Intuitively, the translation guarantees that the value of the final program counter label in the nested computation coincides with the security level of the translated result (bound to x). Therefore, the first component contains the correct label and it is furthermore at the right security level, because $\mathbf{labelOf}(\cdot)$ protects the projected label with the label itself in λ^{dFG}. Primitive $\mathbf{labelOf}(e)$ simply projects the first component of the pair that encodes the labeled value in λ^{dFG} as explained above. Lastly, $\mathbf{getLabel}$ in λ^{dCG} maps directly to $\mathbf{getLabel}$ in λ^{dFG}—rule [GetLabel] in λ^{dCG} simply returns the program counter label and does not raise its value, so it corresponds exactly to rule [GetLabel] in λ^{dFG}, which returns label pc at security level pc. Similarly, $\mathbf{taint}(e)$ translates to $\mathbf{taint}([\![e]\!], ())$, since rule [Taint] in λ^{dCG} taints the program counter with the label that e evaluates to, say ℓ and returns unit with program counter label equal to $pc \sqcup \ell$, which corresponds to the result of the translated program, i.e., $()^{pc \sqcup \ell}$.

References. Figure 16 shows the translation of primitives that access the store via references. Since λ^{dCG}'s rule [New] in Figure 8 creates a new reference

[19] Syntax $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ where x is free in e_2 is a shorthand for $(\lambda x.e_2)\ e_1$.

labeled with the label of the argument (which must be a labeled value), the translation converts $\mathbf{new}(e)$ to an expression that first binds $[\![e]\!]$ to x and then creates a new reference with the same content as the source, i.e., $\mathbf{snd}(x)$, but tainted with the label in x, i.e., $\mathbf{fst}(x)$. Notice that the use of $\mathbf{taint}(\cdot)$ is essential to ensure that λ^{dFG}'s rule [New] in Figure 4 assigns the correct label to the new reference. Due to its *fine-grained* precision, λ^{dFG} might have labeled the content with a different label that is less sensitive than the explicit label that *coarsely* approximates the security level in λ^{dCG}. In contrast, updating a reference does not require any tainting—both λ^{dFG} and λ^{dCG} accept values less sensitive than the reference in rule [Write]. Thus, the translation $e_1 := e_2$ simply updates the translated reference with the content of the labeled value projected from the translated pair, hence $[\![e_1 := e_2]\!]$ is $[\![e_1]\!] := \mathbf{snd}([\![e_2]\!])$. The translation of the primitives that read and query the label of a reference is trivial.

5.1 Cross-Language Equivalence Relation

When a λ^{dCG} program is translated to λ^{dFG} via the program translation described above, the λ^{dFG} result contains strictly more information than the original λ^{dCG} result. This happens because the semantics of λ^{dFG} tracks flows of information at fine granularity, in contrast with λ^{dCG}, which instead coarsely approximates the security level of all values in scope of a computation with the program counter label. When translating a λ^{dCG} program, we can capture this condition precisely for input values θ by *homogeneously* tagging all standard (unlabeled) values with the initial program counter label, i.e., $[\![\theta]\!]^{pc}$. However, a λ^{dCG} program handles, creates and mixes unlabeled data that originated at different security levels at run-time, e.g., when a secret is unlabeled and combined with previously public (unlabeled) data. Crucially, when the translated program executes, the fine-grained semantics of λ^{dFG} tracks those flows of information precisely and thus new labels appear (these labels do not correspond to the label of any labeled value in the source value nor to the program counter label). Intuitively, this implies that the λ^{dFG} result will *not* be homogeneously labeled with the final program counter label of the λ^{dCG} computation, i.e., if a λ^{dCG} program terminates with value v and program counter label pc', the translated λ^{dFG} program does *not* necessarily result in $[\![v]\!]^{pc'}$.

Example. Consider the λ^{dCG} program $\langle \Sigma, L, \mathbf{taint}(H); \mathbf{return}(x)\rangle \Downarrow^{x\mapsto \mathbf{true}} \langle \Sigma, H, \mathbf{true}\rangle$, which returns $\mathbf{true} = \mathbf{inl}()$ and the store Σ unchanged, after tainting the program counter label with H. Let e be the expression obtained by applying the program translation from Figure 15 to the example program:

$$
\begin{array}{l}
e = \lambda\_. \\
\quad \mathbf{let}\ y = \mathbf{taint}(H, ())\ \mathbf{in} \\
\quad\quad \mathbf{taint}(\mathbf{labelOf}(y), x)
\end{array}
$$

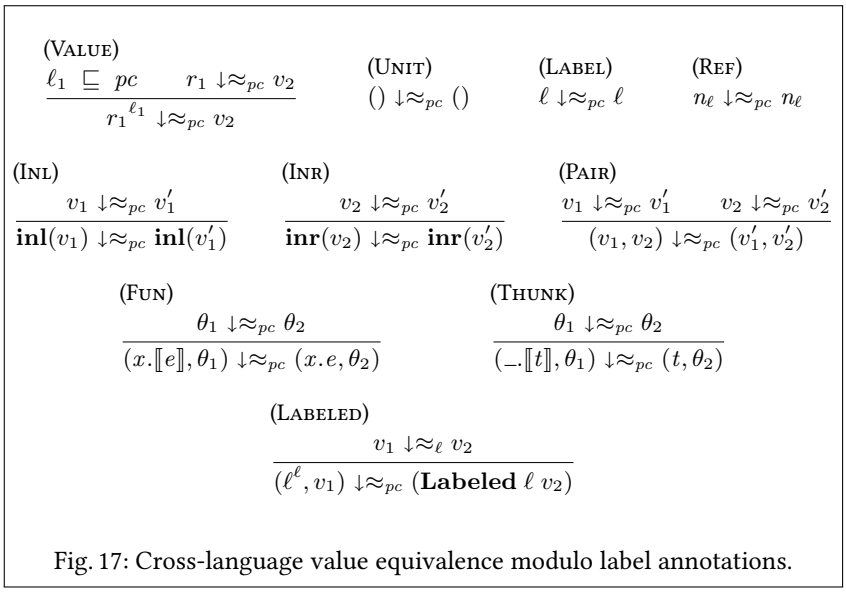


Fig. 17: Cross-language value equivalence modulo label annotations.

Interestingly, when we force the program e and execute it starting from program counter label equal to L, and an input environment translated according to the initial program counter label (L in this case), i.e., $x{\mapsto}[\![\mathbf{true}]\!]^L = \mathbf{inl}(()^L)^L = \mathbf{true}^L$, we do *not* obtain the translated result homogeneously labeled with H:

$$\begin{aligned}\langle [\![\Sigma]\!], e\ ()\rangle \Downarrow_L^{x\mapsto \mathbf{true}^L} \langle [\![\Sigma]\!], \mathbf{true}^H\rangle \quad &= \\ \langle [\![\Sigma]\!], \mathbf{inl}(()^L)^H\rangle \quad &\neq \\ \langle [\![\Sigma]\!], \mathbf{inl}(()^H)^H\rangle \quad &= \\ \langle [\![\Sigma]\!], [\![\mathbf{true}]\!]^H\rangle &\end{aligned}$$

In particular, λ^{dFG} preserves the public label tag on data nested inside the left injection, i.e., $()^L$ in $\mathbf{inl}(()^L)^H$ above. This happens because λ^{dFG}'s rule [Var] taints only the *outer* label of the value $\mathbf{true}^L$ when it looks up variable x in program counter label H.

Solution. In order to recover a notion of semantics preservation, we introduce a key contribution of this work, a *cross-language* binary relation that associates values of the two calculi that, in the scope of a computation at a given security level, are semantically equivalent up to the extra annotations present in the λ^{dFG} value.[20] Technically, we use this equivalence in the semantics preservation theorem in Section 5.2, which *existentially* quantifies over the result of the

[20] This relation is conceptually similar to the logical relation developed by [121] for their translations with *static* IFC enforcement, but technically different in the treatment of labeled values.

translated λ^{dFG} program, but guarantees that it is semantically equivalent to the result obtained in the source program.

Concretely, for a λ^{dFG} value v_1 and a λ^{dCG} value v_2, we write $v_1 \downarrow\!\approx_{pc} v_2$ if the label annotations (including those nested inside compound values) of v_1 are no more sensitive than label pc and its raw value corresponds to v_2. Figure 17 formalizes this intuition by means of two mutually inductive relations, one for λ^{dFG} values and one for λ^{dFG} raw values. In particular, rule [Value] relates λ^{dFG} value ${r_1}^{\ell_1}$ and λ^{dCG} value v_2 at security level pc if the label annotation on the raw value r_1 flows to the program counter label, i.e., $\ell_1 \sqsubseteq pc$, and if the raw value is in relation with the standard value, i.e., $r_1 \downarrow\!\approx_{pc} v_2$. The relation between raw values and standard values relates only semantically equivalent values, i.e., it is syntactic equality for ground types ([Unit,Label,Ref]), requires the same injection for values of the sum type ([Inl,Inr]) and requires the components to related for pairs ([Pair]).

Rules [Fun] (resp. [Thunk]) relates function (resp. thunk) closures only when environments are related pointwise, i.e., $\theta_1 \downarrow\!\approx_{pc} \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x.\theta_1(x) \downarrow\!\approx_{pc} \theta_2(x)$, and the λ^{dFG} function body $x.[\![e]\!]$ (resp. thunk body $\_.[\![t]\!]$) is obtained from the λ^{dCG} function body e (resp. thunk t) via the program translation defined above. Lastly, rule [Labeled] relates a λ^{dCG} labeled value $\mathbf{Labeled}\ \ell\ v_1$ to a pair (ℓ^{ℓ}, v_2), consisting of the label ℓ protected by itself in the first component and value v_2 related with the content v_1 at security level ℓ ($v_1 \downarrow\!\approx_{\ell} v_2$) in the second component. This rule follows the principle of **LIO** that for explicitly labeled values, the label annotation represents an upper bound on the sensitivity of the content. Stores are related pointwise, i.e., $\Sigma_1 \downarrow\!\approx \Sigma_2$ iff $\Sigma_1(\ell) \downarrow\!\approx \Sigma_2(\ell)$ for $\ell \in \mathscr{L}$, and ℓ-labeled memories relate their contents respectively at security level ℓ, i.e., $[\,] \downarrow\!\approx [\,]$ and $(r_1 : M_1) \downarrow\!\approx (r_2 : M_2)$ iff $r_1 \downarrow\!\approx_{\ell} r_2$ and $M_1 \downarrow\!\approx M_2$ for λ^{dFG} and λ^{dCG} memories $M_1, M_2 : Memory\ \ell$. Lastly, we lift the relation to initial and final configurations.

Definition 1 (Equivalence of Configurations)
For all initial and final configurations:

- $\langle \Sigma_1, [\![e]\!]() \rangle \downarrow\!\approx \langle \Sigma_2, pc, e \rangle$ *iff* $\Sigma_1 \downarrow\!\approx \Sigma_2$,
- $\langle \Sigma_1, [\![t]\!] \rangle \downarrow\!\approx \langle \Sigma_2, pc, t \rangle$ *iff* $\Sigma_1 \downarrow\!\approx \Sigma_2$,
- $\langle \Sigma_1, r^{pc} \rangle \downarrow\!\approx \langle \Sigma_2, pc, v \rangle$ *iff* $\Sigma_1 \downarrow\!\approx \Sigma_2$ *and* $r \downarrow\!\approx_{pc} v$.

For initial configurations, the relation requires the λ^{dFG} code to be obtained from the λ^{dCG} expression (resp. thunk) via the program translation function $[\![\cdot]\!]$ defined above (similar to rules [Fun] and [Thunk] in Figure 17). Furthermore, in the first case (expressions), the relation additionally forces the translated suspension $[\![e]\!]$ by applying it to $()$, so that when the λ^{dFG} security monitor executes the translated program, it obtains the result that corresponds to the λ^{dCG} monadic program e. The third definition relates final configurations whenever the stores are related and the security level of the final λ^{dFG}

result corresponds to the program counter label pc of the final λ^{dCG} configuration, and the final λ^{dCG} result corresponds to the λ^{dFG} result up to extra annotations at security level pc, i.e., $r \downarrow\approx_{pc} v$.

Before showing semantics preservation, we prove some basic properties of the equivalence that will be useful later. The following property allows instantiating the semantics preservation theorem with the λ^{dCG} initial configuration. The translation for initial configurations is per-component, i.e., $[\![\langle \Sigma, pc, t\rangle]\!] = \langle [\![\Sigma]\!], [\![t]\!]\rangle$ and forcing for suspensions, i.e., $[\![\langle \Sigma, pc, e\rangle]\!] = \langle [\![\Sigma]\!], [\![e]\!]()\rangle$, pointwise for stores, i.e., $[\![\Sigma]\!] = \lambda\ell.[\![\Sigma(\ell)]\!]$, and memories, i.e., $[\![[\,]]\!] = [\,]$ and $[\![v : M]\!] = [\![v]\!]^{\ell} : [\![M]\!]$ for ℓ-labeled memory M.

Property 3 (Reflexivity) *For all λ^{dCG} initial configurations c, $[\![c]\!] \downarrow\approx c$.*

Proof. The proof is by induction and relies on analogous properties for all syntactic categories: for stores, $[\![\Sigma]\!] \downarrow\approx \Sigma$, for memories, $[\![M]\!] \downarrow\approx M$, for environments $[\![\theta]\!]^{pc} \downarrow\approx_{pc} \theta$, for values $[\![v]\!]^{pc} \downarrow\approx_{pc} v$, for any label pc.

The next property guarantees that values and environments remain in the relation when the program counter label rises.

Property 4 (Weakening) *For all labels pc and pc' such that $pc \sqsubseteq pc'$, and for all λ^{dFG} raw values r_1, values v_1 and environments θ_1, and λ^{dCG} values v_2 and environments θ_2:*

- *If $r_1 \downarrow\approx_{pc} v_2$ then $r_1 \downarrow\approx_{pc'} v_2$*
- *If $v_1 \downarrow\approx_{pc} v_2$ then $v_1 \downarrow\approx_{pc'} v_2$*
- *If $\theta_1 \downarrow\approx_{pc} \theta_2$ then $\theta_1 \downarrow\approx_{pc'} \theta_2$*

Proof. By mutual induction on the cross-language equivalence relation.

5.2 Correctness

With the help of the cross-language relation defined above, we can now state and prove that the λ^{dCG}-to-λ^{dFG} translation is correct, i.e., it satisfies a semantics-preservation theorem analogous to that proved for the translation in the opposite direction. At a high level, the theorem ensures that the translation preserves the meaning of a secure terminating λ^{dCG} program when executed under λ^{dFG} semantics, with the same program counter label and translated input values. Since the translated λ^{dFG} program computes strictly more information than the original λ^{dCG} program, the theorem existentially quantify over the λ^{dFG} result, but insists that it is semantically equivalent to the original λ^{dCG} result at a security level equal to the final value of the program counter label, using the cross-language relation just defined.

We start by proving that the program translation preserves typing.

Lemma 4 (Type Preservation). *If $\Gamma \vdash e : \tau$ then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$.*

Proof. By straightforward induction on the typing judgment.

Next, we prove semantics preservation of λ^{dCG} pure reductions. Since these reductions do not perform any security-relevant operation (they do not read or write state), they can be executed with *any* program counter label in λ^{dFG} and do not change the state in λ^{dFG}.

Lemma 5 ($[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ preserves Pure Semantics).
If $e \Downarrow^{\theta} v$ then for any program counter label pc, λ^{dFG} store Σ, environment θ' such that $\theta' \downarrow\approx_{pc} \theta$, there exists a raw value r, such that $\langle \Sigma, [\![e]\!] \rangle \Downarrow^{\theta'}_{pc} \langle \Sigma, r^{pc} \rangle$ and $r \downarrow\approx_{pc} v$.

Proof. By induction on the given evaluation derivation and using basic properties of the lattice.

Notice that the lemma holds for *any* input target environment θ' in relation with the source environment θ at security level pc rather than just for the translated environment $[\![\theta]\!]^{pc}$. Intuitively, we needed to generalize the lemma so that the induction principle is strong enough to discharge cases where (i) we need to prove reductions that use an existentially quantified environment, e.g., [App] and (ii) when some intermediate result at a security level other than pc gets added to the environment, so the environment is no longer homogenously labeled with pc. While the second condition does not arise in pure reductions, it does occur in the reduction of monadic expressions considered in the following theorem.

Theorem 5 ($[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ preserves Impure Semantics)

- *Let $c_2 = \langle \Sigma, pc, t \rangle$ be an initial λ^{dCG} configuration. If $c_2 \Downarrow^{\theta_2} c_2'$, then for all λ^{dFG} environments θ_1 and initial configurations c_1 such that $\theta_1 \downarrow\approx_{pc} \theta_2$ and $c_1 \downarrow\approx c_2$, there exists a final configuration c_1', such that $c_1 \Downarrow^{\theta_1}_{pc} c_1'$ and $c_1' \downarrow\approx c_2'$.*
- *Let $c_2 = \langle \Sigma, pc, e \rangle$ be an initial λ^{dCG} configuration. If $c_2 \Downarrow^{\theta_2} c_2'$, then for all λ^{dFG} environments θ_1 and initial configurations c_1 such that $\theta_1 \downarrow\approx_{pc} \theta_2$ and $c_1 \downarrow\approx c_2$, there exists a final configuration c_1', such that $c_1 \Downarrow^{\theta_1}_{pc} c_1'$ and $c_1' \downarrow\approx c_2'$.*

Proof (Sketch). By mutual induction on the given derivations, using Lemma 5 for pure reductions and Properties 2 and 4 in cases [Bind, ToLabeled, Unlabel, Read], basic properties of the lattice and of the translation function (for operations on the store).

We finally instantiate the semantics-preservation theorem with the translation of the input values and the initial stores at security level pc.

Corollary 1 (Correctness)
Let $c_2 = \langle \Sigma, pc, e \rangle$, if $c_2 \Downarrow^{\theta} c_2'$, then there exists a final λ^{dFG} configuration c_1' such that $[\![c_2]\!] \Downarrow^{[\![\theta]\!]^{pc}}_{pc} c_1'$ and $c_1' \downarrow\approx c_2'$.

Proof. By Property 3 and Theorem 5.

Notice that the flow-sensitive program counter of the source λ^{dCG} program gets encoded in the security level of the result of the λ^{dFG} translated program. For example, if $\langle \Sigma_2, pc, e\rangle \Downarrow^{\theta} \langle \Sigma_2', pc', v\rangle$ then, by Corollary 1 and unrolling Definition 1, there exists a raw value r at security level pc' and a store Σ_1', such that $\langle [\![\Sigma_2]\!], [\![e]\!]()\rangle \Downarrow_{pc}^{[\![\theta]\!]^{pc}} \langle \Sigma_1', r^{pc'}\rangle$, $r \downarrow\approx_{pc'} v$ and $\Sigma_1' \downarrow\approx \Sigma_2'$.

Recovery of non-interference. Similarly to our presentation of Theorem 4 for the translation in the opposite direction, we conclude this section with a sanity check—recovering a proof of termination-insensitive non-interference (*TINI*) for λ^{dCG} through the program translation defined above, semantics preservation (Corollary 1), λ^{dFG} non-interference (Theorem 1), together with a property that the translation preserves L-equivalence as well (Lemmas 6, 7 and 8). By reproving non-interference of the source language from the target language, we show that our program translation is authentic.

The following lemma ensures that the translation of initial configurations preserves L-equivalence.

Lemma 6. *If $c_1 \approx_L c_2$, then $[\![c_1]\!] \approx_L [\![c_2]\!]$.*

Proof. By induction on the L-equivalence judgment and proving similar lemmas for values, i.e., if $v_1 \approx_L v_2$ then $[\![v_1]\!]^{pc} \approx_L [\![v_2]\!]^{pc}$, for environments, i.e., if $\theta_1 \approx_L \theta_2$ then $[\![\theta_1]\!]^{pc} \approx_L [\![\theta_2]\!]^{pc}$, for any label pc, for memories, i.e., if $M_1 \approx_L M_2$ then $[\![M_1]\!] \approx_L [\![M_2]\!]$, and for stores, i.e., if $\Sigma_1 \approx_L \Sigma_2$ then $[\![\Sigma_1]\!] \approx_L [\![\Sigma_2]\!]$.

The following lemmas recovers λ^{dCG} L-equivalence from λ^{dFG} L-equivalence using the cross-language equivalence relation for all the syntactic categories.

Lemma 7. *For all public program counter labels $pc \sqsubseteq L$, for all λ^{dFG} values v_1, v_2, raw values r_1, r_2, environments θ_1, θ_2, memories M_1, M_2, stores Σ_1, Σ_2, and corresponding λ^{dCG} values v_1', v_2' and environments θ_1', θ_2', memories M_1', M_2', stores Σ_1', Σ_2':*

- *If $v_1 \approx_L v_2$, $v_1 \downarrow\approx_{pc} v_1'$ and $v_2 \downarrow\approx_{pc} v_2'$, then $v_1' \approx_L v_2'$,*
- *If $r_1 \approx_L r_2$, $r_1 \downarrow\approx_{pc} v_1'$ and $r_2 \downarrow\approx_{pc} v_2'$, then $v_1' \approx_L v_2'$,*
- *If $\theta_1 \approx_L \theta_2$, $\theta_1 \downarrow\approx_{pc} \theta_1'$ and $\theta_2 \downarrow\approx_{pc} \theta_2'$, then $\theta_1' \approx_L \theta_2'$,*
- *If $M_1 \approx_L M_2$, $M_1 \downarrow\approx M_1'$ and $M_2 \downarrow\approx M_2'$, then $M_1' \approx_L M_2'$,*
- *If $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1 \downarrow\approx \Sigma_1'$ and $\Sigma_2 \downarrow\approx \Sigma_2'$, then $\Sigma_1' \approx_L \Sigma_2'$.*

Proof. The lemmas are proved mutually, by induction on the L-equivalence relation and the cross-language equivalence relations and using injectivity of the translation function $[\![\cdot]\!]$ for closure values.[21]

The next lemma lifts the previous lemma final configurations.

[21] Technically, the function $[\![\cdot]\!]$ presented in Section 5 is not injective. For example, consider the type translation function from Figure 14a: $[\![\textbf{Labeled unit}]\!] = \mathscr{L} \times \textbf{unit} = [\![\mathscr{L} \times \textbf{unit}]\!]$ but $\textbf{Labeled unit} \neq \mathscr{L} \times \textbf{unit}$, and $[\![\textbf{LIO unit}]\!] =$

Lemma 8. *Let c_1 and c_2 be λ^{dFG} final configurations, let c_1' and c_2' be λ^{dCG} final configurations. If $c_1 \approx_L c_2$, $c_1 \downarrow\approx c_1'$ and $c_2 \downarrow\approx c_2'$, then $c_1' \approx_L c_2'$.*

Proof. Let $c_1 = \langle \Sigma_1, v_1\rangle$, $c_2 = \langle \Sigma_2, v_2\rangle$, $c_1' = \langle \Sigma_1', pc_1, v_1'\rangle$, $c_2' = \langle \Sigma_2', pc_2, v_2'\rangle$. From L-equivalence of λ^{dFG} final configurations, it follows L-equivalence for the stores and the values, i.e., $\Sigma_1 \approx_L \Sigma_2$ and $v_1 \approx_L v_2$ from $c_1 \approx_L c_2$ (Section 2.2). Similarly, from cross-language equivalence of final λ^{dFG} and λ^{dCG} configurations, it follows cross-language equivalence of their components, i.e., respectively $\Sigma_1 \downarrow\approx \Sigma_1'$ and $v_1 \downarrow\approx_{pc_1} v_1'$ from $c_1 \downarrow\approx c_2$, and $\Sigma_2 \downarrow\approx \Sigma_2'$ and $v_2 \downarrow\approx_{pc_2} v_2'$ from $c_2 \downarrow\approx c_2'$ (Definition 1). First, we show that the λ^{dCG} stores are L-equivalent, i.e., $\Sigma_1' \approx_L \Sigma_2'$ by Lemma 7 for stores, then two cases follow by case split on $v_1 \approx_L v_2$. Either (i) both label annotations on the values are not observable ([Value$_H$]), then the program counter labels are also not observable ($pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$ from $v_1 \downarrow\approx_{pc_1} v_1'$ and $v_2 \downarrow\approx_{pc_2} v_2'$) and $c_1' \approx_L c_2'$ by rule [Pc$_H$] or (ii) the label annotations are equal and observable by the attacker ([Value$_L$]), i.e., $pc_1 \equiv pc_2 \sqsubseteq L$, then $v_1' \approx_L v_2'$ by Lemma 7 for values and $c_1' \approx_L c_2'$ by rule [Pc$_L$].

Theorem 6 (λ^{dCG}-TINI via $[\![\cdot]\!]$)
If $c_1 \Downarrow^{\theta_1} c_1'$, $c_2 \Downarrow^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ and $c_1 \approx_L c_2$, then $c_1' \approx_L c_2'$.

Proof. First, we apply the translation $[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ to the initial configurations c_1 and c_2 and the respective environments θ_1 and θ_2. Let pc be the initial program counter label common to configurations c_1 and c_2 (it is the same because $c_1 \approx_L c_2$). Corollary 1 (Correctness) then ensures that there exist two λ^{dFG} configurations c_1'' and c_2'', such that $[\![c_1]\!] \Downarrow_{pc}^{[\![\theta_1]\!]^{pc}} c_1''$ and $c_1'' \downarrow\approx c_1'$, and $[\![c_2]\!] \Downarrow_{pc}^{[\![\theta_2]\!]^{pc}} c_2''$ and $c_2'' \downarrow\approx c_2'$. We then lift L-equivalence of source configurations and environments to L-equivalence in the target language via Lemma 6, i.e., $[\![\theta_1]\!]^{pc} \approx_L [\![\theta_2]\!]^{pc}$ and $[\![c_1]\!] \approx_L [\![c_2]\!]$, and apply Theorem 1 ($\lambda^{dFG}$-*TINI*) to the reductions i.e., $[\![c_1]\!] \Downarrow_{pc}^{[\![\theta_1]\!]^{pc}} c_1''$ and $[\![c_2]\!] \Downarrow_{pc}^{[\![\theta_2]\!]^{pc}} c_2''$, which gives L-equivalence of the resulting configurations, i.e., $c_1'' \approx_L c_2''$. Then, we apply Lemma 8 to $c_1'' \approx_L c_2''$, $c_1'' \downarrow\approx c_1'$, and $c_2'' \downarrow\approx c_2'$, and recover L-equivalence for the source configurations, i.e., $c_1' \approx_L c_2'$.

6 Related work

Systematic study of the relative expressiveness of fine- and coarse-grained information flow control (IFC) systems has started only recently. [120] initiated this study in the context of *static* coarse- and fine-grained IFC, enforced via type

---

$\mathbf{unit} \to \mathbf{unit} = [\![\mathbf{unit} \to \mathbf{unit}]\!]$ but $\mathbf{LIO}\ \mathbf{unit} \neq \mathbf{unit} \to \mathbf{unit}$. We make the translation injective by (i) adding a wrapper type $\mathbf{Id}\ \tau$ to λ^{dFG}, together with constructor $\mathbf{Id}(e)$, a deconstructor $\mathbf{unId}(e)$ and raw value $\mathbf{Id}(v)$, and (ii) tagging security-relevant types and terms with the wrapper, i.e., $[\![\mathbf{Labeled}\ \tau]\!] = \mathbf{Id}\ (\mathscr{L} \times [\![\tau]\!])$ and $\mathbf{LIO}\ \tau = \mathbf{Id}\ \mathbf{unit} \to [\![\tau]\!]$. Adapting the translations in both directions is tedious but straightforward; we refer the interested reader to our mechanized proofs for details.

systems. In more recent work, [121] show that a fine-grained IFC type system, which they call FG, and two variants of a coarse-grained IFC type system, which they call CG, are equally expressive. Their approach is based on type-directed translations, which are type- and semantics-preserving. For proofs, they develop logical relations models of FG and the two variants of CG, as well as cross-language logical relations. Our work and some of our techniques are directly inspired by their work, but we examine *dynamic* IFC systems based on runtime monitors. As a result, our technical development is completely different. In particular, in our work we handle label introspection, which has no counterpart in the earlier work on static IFC systems, and which also requires significant care in translations. Our dynamic setting also necessitated the use of tainting operators in both the fine-grained and the coarse-grained systems.

Our coarse-grained system λ^{dCG} is the dynamic analogue of the second variant of [121]'s CG type system. This variant is described only briefly in their paper (in Section 4, paragraph "Original HLIO") but covered extensively in Part-II of the paper's appendix. [121] argue that translating their fine-grained system FG to this variant of CG is very difficult and requires significant use of parametric label polymorphism. The astute reader may wonder why we do not encounter the same difficulty in translating our fine-grained system λ^{dFG} to λ^{dCG}. The reason for this is that our fine-grained system λ^{dFG} is not a direct dynamic analogue of [121]'s FG. In λ^{dFG}, a value constructed in a context with program counter label pc automatically receives the security label pc. In contrast, in [121]'s FG, all introduction rules create values (statically) labeled $\bot$. Hence, leaving aside the static-vs-dynamic difference, FG's labels are more precise than λ^{dFG}'s, and this makes [121]'s FG to CG translation more difficult than our λ^{dFG} to λ^{dCG} translation. In fact, in earlier work, [120] introduced a different type system called FG^{-}, a static analogue of λ^{dFG} that labels all constructed values with pc (statically), and noted that translating it to the second variant of CG is much easier (in the static setting).

Coarse-grained dynamic IFC systems are prevalent in security research in operating systems [39,69,166]. These ideas have also been successfully applied to other domains, e.g., the web [18,45,69,146], mobile applications [64,101], and IoT [43]. LIO is a domain-specific language embedded in Haskell that rephrases OS-like IFC enforcement into a language-based setting [141,145]. [55] introduce a general framework for coarse-grained IFC in any programming language in which external effects can be controlled. Laminar [122] unifies mechanisms for IFC in programming languages and operating systems, resulting in a mix of dynamic fine- and coarse-grained enforcement.

In general, dynamic fine-grained IFC systems often do not support label introspection. LIO [144, 145] and Breeze [59] are notable exceptions. Breeze is conceptually similar to our λ^{dFG} except for the $\mathbf{taint}(\cdot)$ primitive. Different from our λ^{dFG}, there are dynamic fine-grained IFC systems in which labels of references are flow-sensitive [7, 8, 22, 51]. This design choice, however, allows label changes to be exploited as a covert channel for information leaks [7,8,127].

There are many approaches to preventing such leaks—from using static analysis techniques [128], to disallowing label upgrades depending on sensitive data (i.e., no-sensitive-upgrades [7, 165]), to avoiding branching on data whose labels have been upgraded (i.e., permissive-upgrades [8]). Extending our results to a fine-grained dynamic IFC system with flow-sensitive references is an interesting direction for future work.

7 Conclusion

We formally established a connection between dynamic fine- and coarse-grained enforcement for IFC, showing that both are equally expressive under reasonable assumptions. Indeed, this work provides a systematic way to bridging the gap between a wide range of dynamic IFC techniques often proposed by the programming languages (fine-grained) and operating systems (coarse-grained) communities. As consequence, this allows future designs of dynamic IFC to choose a coarse-grained approach, which is easier to implement and use, without giving up on the precision of fine-grained IFC.

References

1. Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.
2. Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.
3. Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Computer Languages, Systems & Structures*, 33(2):35–59, 2007.
4. Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Annual Network & Distributed System Security Symposium*. Internet Society, 2015.
5. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 159–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
6. Niklas Broberg, Bart Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 217–232, Berlin, Heidelberg, 2013. Springer-Verlag.
7. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.
8. Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.
9. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and

event processes in the Asbestos operating system. In *ACM Symposium on Operating Systems Principles*, SOSP. ACM, 2005.
10. Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1-3):35–75, December 1991.
11. Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium*, pages 531–548, 2016.
12. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.
13. J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.
14. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing*. ACM, 2014.
15. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 11–31, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
16. C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
17. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, pages 170–178, Berlin, Heidelberg, 2012. Springer-Verlag.
18. Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android. In *Computer Security – ESORICS 2013*, pages 775–792, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
19. Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP. ACM, 2007.
20. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.
21. Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *USENIX Security Symposium*, pages 1119–1136, 2016.
22. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.
23. Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. Type systems for information flow control: The question of granularity. *ACM SIGLOG News*, 4(1):6–21, February 2017.
24. Vineet Rajani and Deepak Garg. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *Proc. of the IEEE Computer Security Foundations Symp.*, CSF '18. IEEE Computer Society, 2018.
25. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2009.

26. Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 280–288, New York, NY, USA, 2015. ACM.
27. Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 13–24, New York, NY, USA, 2008. ACM.
28. Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.
29. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.
30. Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1617–1634, New York, NY, USA, 2018. ACM.
31. Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.
32. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27, 2017.
33. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.
34. Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014.
35. Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 187–202, Washington, DC, USA, 2007. IEEE Computer Society.
36. Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 15–28, New York, NY, USA, 2016. ACM.
37. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.
38. Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.
39. Stephan Arthur Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002. AAI3063751.
40. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.

41. Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

Paper VI

Based on

Towards Foundations for Parallel Information Flow Control Runtime Systems,

by Marco Vassena, Gary Soeller, Peter Amidon,

Matthew Chan, John Renner and Deian Stefan,

8th International Conference on Principles of Security and Trust.

CHAPTER

FIVE

TOWARDS FOUNDATIONS FOR PARALLEL IFC RUNTIME SYSTEMS

Abstract. We present the foundations for a new dynamic information flow control (IFC) parallel runtime system, $\mathbf{LIO}_{\mathrm{PAR}}$. To our knowledge, $\mathbf{LIO}_{\mathrm{PAR}}$ is the first dynamic language-level IFC system to (1) support deterministic parallel thread execution and (2) eliminate both internal- and external-timing covert channels that exploit the runtime system. Most existing IFC systems are vulnerable to external timing attacks because they are built atop vanilla runtime systems that do not account for security—these runtime systems allocate and reclaim shared resources, e.g., CPU-time and memory, *fairly* between threads at different security levels. While such attacks have largely been ignored—or, at best, mitigated—we demonstrate that extending IFC systems with parallelism leads to the *internalization* of these attacks. Our IFC runtime system design addresses these concerns by hierarchically managing resources—both CPU-time and memory—and making resource allocation and reclamation explicit at the language-level. We prove that $\mathbf{LIO}_{\mathrm{PAR}}$ is secure, i.e., it satisfies timing-sensitive non-interference, even when exposing clock and heap-statistics APIs.

1 Introduction

Language-level dynamic information flow control (IFC) is a promising approach to building secure software systems. With IFC, developers specify application-specific, data-dependent security policies. The language-level IFC system—often implemented as a library or as part of a language runtime system—then enforces these policies automatically, by tracking and restricting the flow of information throughout the application. In doing so, IFC can ensure that different application components—even when buggy or malicious—cannot violate data confidentiality or integrity.

The key to making language-level IFC practical lies in designing real-world programming language features and abstractions without giving up on security. Unfortunately, many practical language features are at odds with security. For example, even exposing language features as simple as `if`-statements can expose users to timing attacks [100, 158]. Researchers have made significant strides towards addressing these challenges—many IFC systems now support real-world features and abstractions safely [43, 51, 59, 80, 101, 122, 123, 141, 144, 151, 152, 154, 163, 164]. To the best of our knowledge, though, no existing language-level dynamic IFC supports parallelism. Yet, many applications rely on parallel thread execution. For example, modern Web applications typically handle user requests in parallel, on multiple CPU cores, taking advantage of modern hardware. Web applications built atop state-of-the-art dynamic IFC Web frameworks (e.g., Jacqueline [163], Hails [45, 46], and LMonad [106]), unfortunately, do not handle user requests in parallel—the language-level IFC systems that underlie them (e.g., Jeeves [164] and LIO [141]) do not support parallel thread execution.

In this paper we show that extending most existing IFC systems—even concurrent IFC systems such as LIO—with parallelism is unsafe. The key insight is that most IFC systems *do not* prevent sensitive computations from affecting public computations; they simply prevent public computations from *observing* such sensitive effects. In the sequential and concurrent setting, such effects are only observable to attackers *external* to the program and thus outside the scope of most IFC systems. However, when computations execute in parallel, they are essentially external to one another and thus do not require an observer external to the system—they can observe such effects internally.

Consider a program consisting of three concurrent threads: two public threads—p_0 and p_1—and a secret thread—s_0. On a single core, language-level IFC can ensure that p_0 and p_1 do not learn anything secret by, for example, disallowing them from observing the return values (or lack thereof) of the secret thread. Systems such as LIO are careful to ensure that public threads cannot learn secrets even indirectly, e.g., via covert channels that abuse the runtime system scheduler. In contrast, secret threads can leak information to an external observer that monitors public events (e.g., messages from public threads) by influencing the behavior of the public threads. For example, s_0 can terminate (or not) based on a secret and thus affect the amount of time p_0 and p_1 spend executing on the CPU—if s_0 terminated, the runtime allots the whole CPU to public threads, otherwise it only allots, say, two thirds of the CPU to the public threads; this allows an external attacker to trivially infer the secret (e.g., by measuring the rate of messages written to a public channel). Unfortunately, such *external timing attacks* manifest *internally* to the program when threads execute in parallel, on multiple cores. Suppose, for example, that p_0 and s_0 are co-located on a core and run in parallel to p_1. By terminating early (or not) based on a secret, s_0 affects the CPU time allotted to p_0, which can be measured by p_1. For example, p_1 can count the number of messages sent from p_0 on a

public channel—the number of p_0 writes indirectly leaks whether or not s_0 terminated.

We demonstrate that such attacks are feasible by building several proof-of-concept programs that exploit the way the runtime system allocate and reclaim *shared* resources to violate LIO's security guarantees. Then, we design a new dynamic parallel language-level IFC runtime system called $\mathbf{LIO}_{\mathrm{PAR}}$, which extends LIO to the parallel setting by changing how *shared* runtime system resources—namely CPU-time and memory—are managed. Ordinary runtime systems (e.g., GHC for LIO) *fairly* balance resources between threads; this means that allocations or reclamations for secret LIO threads directly affect resources available for public LIO threads. In contrast, $\mathbf{LIO}_{\mathrm{PAR}}$ makes resource management *explicit* and *hierarchical*. When allocating new resources on behalf of a thread, the $\mathbf{LIO}_{\mathrm{PAR}}$ runtime does not "fairly" steal resources from all threads. Instead, $\mathbf{LIO}_{\mathrm{PAR}}$ demands that the thread requesting the allocation explicitly gives up a portion of its own resources. Similarly, the runtime does not automatically relinquish the resources of a terminated thread—it requires the parent thread to explicitly reclaim them.

Nevertheless, automatic memory management is an integral component of modern language runtimes—high-level languages (e.g., Haskell and thus LIO) are typically garbage collected, relieving developers from manually reclaiming unused memory. Unfortunately, even if memory is hierarchically partitioned, some garbage collection (GC) algorithms, such as GHC's stop-the-world, may introduce timing covert channels [107]. Inspired by previous work on real-time GCs (e.g., [6, 13, 23, 54, 104, 114]), we equip $\mathbf{LIO}_{\mathrm{PAR}}$ with a per-thread, interruptible garbage collector. This strategy is agnostic to the particular GC algorithm used: our hierarchical runtime system only demands that the GC runs within the memory confines of individual threads and their time budget.

In sum, this paper makes three contributions:

▷ We observe that several external timing attacks *manifest internally* in the presence of parallelism and demonstrate that LIO, when compiled to run on multiple cores, is vulnerable to such attacks (§2).

▷ In response to these attacks, we propose a novel parallel runtime system design that safely manages shared resources by enforcing explicit and *hierarchical* resource allocation and reclamation (§3). To our knowledge, $\mathbf{LIO}_{\mathrm{PAR}}$ is the first parallel language-level dynamic IFC runtime system to address both internal and external timing attacks that abuse the runtime system scheduler, memory allocator, and GC.

▷ We formalize the $\mathbf{LIO}_{\mathrm{PAR}}$ hierarchical runtime system (§4) and prove that it satisfies *timing-sensitive non-interference* (§5); we believe that this is the first general purpose dynamic IFC runtime system to provide such strong guarantees in the parallel setting [158].

We remark that neither our attack nor our defense is tied to LIO or GHC—we focus on LIO because it already supports concurrency. We believe that extending any existing language-level IFC system with parallelism will pose

the same set of challenges—challenges that can be addressed using explicit and hierarchical resource management. Supplemental materials (detailed formal definitions and proofs) can be found in Appendixes A and B while the source code for our attacks can be found in Appendix C.

2 Internal Manifestation of External Timing Attacks

In this section we give a brief overview of LIO and discuss the implications of shared, finite runtime system resources on security. We demonstrate several external timing attacks against LIO that abuse two such resources—the thread scheduler and garbage collector—and show how running LIO threads in parallel internalizes these attacks.

2.1 Overview of Concurrent LIO

At a high level, the goal of an IFC system is to track and restrict the flow of information according to a security policy—almost always a form of *non-interference* [48]. Informally, this policy ensures *confidentiality*, i.e., secret data should not leak to public entities, and *integrity*, i.e., untrusted data should not affect trusted entities.

To this end, LIO tracks the flow of information at a coarse-granularity, by associating *labels* with threads. Implicitly, the thread label classifies all the values in its scope and reflects the sensitivity of the data that it has inspected. Indeed, LIO "raises" the label of a thread to accommodate for reading yet more sensitive data. For example, when a $public$ thread reads $secret$ data, its label is raised to $secret$—this reflects the fact that the rest of the thread computation may depend on sensitive data. Accordingly, LIO uses the thread's *current label* or *program counter label* to restrict its communication. For example, a $secret$ thread can only communicate with other $secret$ threads.

In LIO, developers can express programs that manipulate data of varying sensitivity—for example programs that handle both $public$ and $secret$ data—by forking multiple threads, at run-time, as necessary. However, naively implementing concurrency in an IFC setting is dangerous: concurrency can amplify and internalize the *termination covert channel* [4, 149], for example, by allowing public threads to observe whether or not secret threads terminated. Moreover, concurrency often introduces *internal timing covert channels* wherein secret threads leak information by influencing the scheduling behavior of public threads. Both classes of covert channels are high-bandwidth and easy to exploit.

Stefan et al. [141] were careful to ensure that LIO does not expose these termination and timing covert channels *internally*. LIO ensures that even if secret threads terminate early, loop forever, or otherwise influence the runtime system scheduler, they cannot leak information to public threads. But, secret threads *do* affect public threads with those actions and thus expose timing covert channels *externally*—public threads just cannot detect it. In particular, LIO disallows public threads from (1) directly inspecting the return values (and thus timing and termination behavior) of secret threads, without first raising

their program counter label, and (2) observing runtime system resource usage (e.g., elapsed time or memory availability) that would indirectly leak secrets.

LIO prevents public threads from measuring CPU-time usage directly—LIO does not expose a clock API—and indirectly—threads are scheduled fairly in a round-robin fashion [141]. Similarly, LIO prevents threads from measuring memory usage directly—LIO does not expose APIs for querying heap statistics—and indirectly, through garbage collection cycles (e.g., induced by secret threads) [107]—GHC's stop-the-world GC stops all threads. Like other IFC systems, the security guarantees of LIO are weaker in practice because its formal model does not account for the GC and assumes memory to be infinite [141, 144].

2.2 External Timing Attacks to Runtime Systems

Since secret threads can still influence public threads by abusing the scheduler and GC, LIO is vulnerable to *external timing and termination attacks*, i.e., attacks that leak information to external observers.To illustrate this, we craft several LIO programs consisting of two threads: a public thread p that writes to the external channel observed by the attacker and a secret thread s, which abuses the runtime to influence the throughput of the public thread. The secret thread can leak in many ways, for example, thread s can:

1. *fork bomb*, i.e., fork thousands of secret threads that will be interleaved with p and thus decrease its write throughput;
2. terminate early to relinquish the CPU to p and thus double its write throughput;
3. exhaust all memory to crash the program, and thus stop p from further writing to the channel;
4. force a garbage collection which, because of GHC's stop-the-world GC, will intermittently stop p from writing to the channel.

These attacks abuse the runtime's automatic *allocation* and *reclamation* of shared resources, i.e., CPU time and memory. In particular, attack 1 hinges on the runtime *allocating* CPU time for the new secret threads, thus reducing the CPU time allotted to the public thread. Dually, attack 2 relies on it *reclaiming* the CPU time of terminated threads—it reassigns it to public threads. Similarly, attacks 3 and 4 force the runtime to allocate all the available memory and preemptively reassign CPU time to the GC, respectively.

These attacks are not surprising, but, with the exception of the GC-based attack [107], they are novel in the IFC context. Moreover these attacks are not exhaustive—there are other ways to exploit the runtime system—nor optimized—our implementation leaks sensitive data at a rate of roughly 2bits/second[1]. Nevertheless, they are feasible and—because they abuse the runtime—they are effective against language-level external-timing mitigation

[1] A more assiduous attacker could craft similar attacks that leak at higher bit-rates.

Core c_0		**Core** c_1
Secret Thread (s_0)	Public Thread (p_0)	Public Thread (p_1)

`if secret`
`  then terminate`
`  else forever skip`
	`forever`
	`  (write chan` p_0`)`
		`for [1..`n`]`
		`  write chan` p_1
		$w_s \leftarrow$ `read chan`
		$w_0 \leftarrow$ `count` p_0 w_s
		$w_1 \leftarrow$ `count` p_1 w_s
		`return (`w_0 `<` w_1`)`

Fig. 1: In this attack three threads run in parallel, colluding to leak secret `secret`. The two public threads write to a *public* output channel; the relative number of messages written on the channel by each thread directly leaks the secret (as inferred by p_1). To affect the rate that p_0 can write, s_0 conditionally terminates—which will free up time on core c_0 for p_0 to execute.

techniques, including [141,168]. The attacks are also feasible on other systems—similar attacks that abuse the GC have been demonstrated for both the V8 and JVM runtimes [107].

2.3 Internalizing External Timing Attacks

LIO, like almost all IFC systems, considers external timing out of scope for its attacker model. Unfortunately, when we run LIO threads on multiple cores, in parallel, the allocation and reclamation of resources on behalf of secret threads is indirectly observable by public threads. Unsurprisingly, some of the above external timing attacks manifest internally—a thread running on a parallel core acts as an "external" attacker. To demonstrate the feasibility of such attacks, we describe two variants of the aforementioned scheduler-based attacks which leak sensitive information internally to public threads.

Secret threads can leak information by relinquishing CPU time, which the runtime reclaims and *unsafely* redistributes to public threads running on the same core. Our attack program consists of three threads: two public threads—p_0 and p_1—and a secret thread—s_0. Fig. 1 shows the pseudo-code for this attack. Note that the threads are secure in isolation, but leak the value of $secret$ when executed in parallel, with a round robin scheduler. In particular, threads p_0 and s_0 run concurrently on core c_0 using half of the CPU time each, while p_1 runs in parallel alone on core c_1 using all the CPU time. Both public threads repeatedly write their respective thread IDs to a *public channel*. The secret thread, on the other hand, loops forever or terminates depending on $secret$. Intuitively, when the secret thread terminates, the runtime system redirects its CPU time to p_0, thus both p_1 and p_0 write at the same rate. In converse, when the secret thread does not terminate early, p_0 is scheduled in a round-robin fashion with s_0 on the same core and can thus only write half as fast as p_1. More specifically:

▷ If `secret = true`, thread s_0 terminates and the runtime system assigns all the CPU time of core c_0 to public thread p_0, which then writes at the same

rate as thread p_1 on core c_1. Then, p_0 writes as many times as p_1, which then returns `true`.

▷ If `secret = false`, secret thread s_0 loops and public thread p_0 shares the CPU time on core c_0 with it. Then, p_0 writes messages at roughly half the rate of thread p_1, which writes more often—it has all the CPU time on c_1—and thus returns `false`.[2]

Secret LIO threads can also leak information by allocating many secret threads on a core with public threads—this reduces the CPU-time available to the public threads. For example, using the same setting with three threads from before, the secret thread forks a spinning thread on core c_1 by replacing command `terminate` with command `fork (forever skip)` c_1 in the code of thread s_0 in Fig. 1. Intuitively, if `secret` is `false`, then p_1 writes more often than p_0 before, otherwise the write rate of p_1 decreases—it shares core c_1 with the child thread of s_0—and p_0 writes as often as p_1.

Not all external timing attacks can be internalized, however. In particular, GHC's approach to reclaiming memory via a stop-the-world GC simultaneously stops all threads on *all* cores, thus the relative write rate of public threads remain constant. Interestingly, though, implementing LIO on runtimes (e.g., Node.js as proposed by Heule et al. [55]) with modern parallel garbage collectors that do not always stop the world would internalize the GC-based external timing attacks. Similarly, abusing GHC's memory allocation to exhaust all memory crashes all the program threads and, even though it cannot be internalized, it still results in information leakage.

3 Secure Parallel Runtime System

To address the external and internal timing attacks, we propose a new dynamic IFC runtime system design. Fundamentally, today's runtime systems are vulnerable because they automatically allocate and reclaim resources that are shared across threads of varying sensitivity. However, the automatic allocation and reclamation is not in itself a problem—it is only a problem because the runtime steals (and grants) resources from (and to) differently-labeled threads.

Our runtime system, **LIO**$_{\mathrm{PAR}}$, explicitly partitions CPU-time and memory among threads—each thread has a fixed CPU-time and memory *budget* or *quota*. This allows resource management decisions to be made locally, for each thread, independent of the other threads in the system. For example, the runtime scheduler of **LIO**$_{\mathrm{PAR}}$ relies on CPU-time partitioning to ensure that threads always run for a fixed amount of time, irrespective of the other threads running on the same core. Similarly, in **LIO**$_{\mathrm{PAR}}$, the memory allocator and garbage collector rely on memory partitioning to be able to allocate and collect memory on behalf of a thread without being influenced or otherwise influencing other threads in the system. Furthermore, partitioning resources among threads enables fine-

[2] The attacker needs to empirically find parameter n, so that p_1 writes roughly twice as much as thread p_0 with half CPU time on core c_0.

grained control of resources: **LIO**$_{\mathrm{PAR}}$ exposes secure primitives to (i) measure resource usage (e.g., time and memory) and (ii) elicit garbage collection cycles.

The **LIO**$_{\mathrm{PAR}}$ runtime does not automatically balance resources between threads. Instead, **LIO**$_{\mathrm{PAR}}$ makes resource management explicit at the language level. When forking a new thread, for example, **LIO**$_{\mathrm{PAR}}$ demands that the parent thread give up part of its CPU-time and memory budgets to the children. Indeed, **LIO**$_{\mathrm{PAR}}$ even manages core ownership or *capabilities* that allow threads to fork threads across cores. This approach ensures that allocating new threads does not indirectly leak any information externally or to other threads. Dually, the **LIO**$_{\mathrm{PAR}}$ runtime does not re-purpose unused memory or CPU-time, even when a thread terminates or "dies" abruptly—parent threads must explicitly kill their children when they wish to reclaim their resources.

To ensure that CPU-time and memory can always be reclaimed, **LIO**$_{\mathrm{PAR}}$ allows threads to kill their children anytime. Unsurprisingly, this feature requires restricting the **LIO**$_{\mathrm{PAR}}$ floating-label approach more than that of LIO—**LIO**$_{\mathrm{PAR}}$ threads cannot raise their current label if they have already forked other threads. As a result, in **LIO**$_{\mathrm{PAR}}$ threads form a *hierarchy*—children threads are always at least as sensitive as their parent—and thus it is secure to expose an API to *allocate* and *reclaim* resources.

Attacks Revisited. **LIO**$_{\mathrm{PAR}}$ enforces security against *reclamation-based attacks* because secret threads cannot automatically relinquish their resources. For example, our hierarchical runtime system stops the attack in Fig. 1: even if secret thread s_0 terminates (`secret = true`), the throughput of public thread p_0 remains constant—**LIO**$_{\mathrm{PAR}}$ does not reassign the CPU time of s_0 to p_0, but keeps s_0 spinning until it gets killed. Similarly, **LIO**$_{\mathrm{PAR}}$ protects against *allocation-based attacks* because secret threads cannot steal resources owned by other public threads. For example, the *fork-bomb* variant of the previous attack fails because **LIO**$_{\mathrm{PAR}}$ aborts command `fork (forever skip)` c_1—thread s_0 does not own the core capability c_1—and thus the throughput of p_1 remains the same. In order to substantiate these claims, we first formalize the design of the *hierarchical* runtime system (§4) and establish its security guarantees (§5).

Trust model. This work addresses attacks that exploit runtime system resource management—in particular memory and CPU-time. We do not address attacks that exploit other shared runtime system state (e.g., event loops [155], lazy evaluation [31, 151]), shared operating system state (e.g., file system locks [65], events and I/O [61, 78]), or shared hardware (e.g., caches, buses, pipelines and hardware threads [44, 108]) Though these are valid concerns, they are orthogonal and outside the scope of this paper.

4 Hierarchical Calculus

In this section we present the formal semantics of **LIO**$_{\mathrm{PAR}}$. We model **LIO**$_{\mathrm{PAR}}$ as a security monitor that executes simply typed λ-calculus terms extended with LIO security primitives on an abstract machine in the style of Sestoft [135]. The security monitor reduces secure programs and aborts the execution of leaky programs.

Label	$\ell, pc, cl \in \mathscr{L}$	Params.	$\mu ::= (h, cl)$
Cores	$k \in \{1..\kappa\}, K \in \mathcal{P}(\{1..\kappa\})$	Heap	$\Delta \in Var \rightharpoonup Term$
Thread Id	$n \in \mathbb{N}, N \in \mathcal{P}(\mathbb{N})$	Budgets	$h, b \in \mathbb{N}$
Stack	$S ::= [\,] \mid C : S$	State	$s ::= (\Delta, pc, N \mid t, S)$

$$
\begin{array}{lll}
\text{Type} & \tau & ::= () \mid \tau_1 \to \tau_2 \mid \mathit{Bool} \mid \mathscr{L} \mid \mathit{LIO}\ \tau \mid \mathit{Labeled}\ \tau \\
 & & \quad \mid \mathit{TId} \mid \mathit{Core} \mid \mathcal{P}(\{1..\kappa\}) \mid \mathbb{N} \\
\text{Value} & v & ::= () \mid \lambda x.t \mid \mathit{True} \mid \mathit{False} \mid \ell \mid \mathit{return}\ t \mid \mathit{Labeled}\ \ell\ t^\circ \\
 & & \quad \mid n \mid k \mid K \\
\text{Term} & t & ::= v \mid x \mid t_1\ t_2 \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid t_1 \gg\!= t_2 \mid \mathit{label}\ t_1\ t_2 \\
 & & \quad \mid \mathit{unlabel}\ t \mid \mathit{fork}\ t_1\ t_2\ t_3\ t_4\ t_5 \mid \mathit{spawn}\ t_1\ t_2\ t_3\ t_4\ t_5 \mid \mathit{kill}\ t \\
 & & \quad \mid \mathit{size} \mid \mathit{time} \mid \mathit{wait}\ t \mid \mathit{send}\ t_1\ t_2 \mid \mathit{receive} \\
\text{CTerm} & t^\circ & ::= t \text{ such that } \mathit{fv}(t) = \varnothing \\
\text{Cont.} & C & ::= x \mid \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid \gg\!= t_2 \mid \mathit{label}\ t \mid \mathit{unlabel} \\
 & & \quad \mid \mathit{fork}\ t_1\ t_2\ t_3\ t_4 \mid \mathit{spawn}\ t_1\ t_2\ t_3\ t_4 \mid \mathit{kill} \mid \mathit{send}\ t
\end{array}
$$

(App$_1$)
$$\frac{|\Delta| < \mu.h \qquad \text{fresh}(x)}{(\Delta, pc, N \mid t_1\ t_2, S) \rightsquigarrow_\mu (\Delta[x \mapsto t_2], pc, N \mid t_1, x : S)}$$

(App$_2$)
$$(\Delta, pc, N \mid \lambda y.t, x : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t\ [x\ /\ y], S)$$

(Var)
$$\frac{x \mapsto t \in \Delta}{(\Delta, pc, N \mid x, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t, S)}$$

(Bind$_1$)
$$(\Delta, pc, N \mid t_1 \gg\!= t_2, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \gg\!= t_2 : S)$$

(Bind$_2$)
$$(\Delta, pc, N \mid \mathit{return}\ t_1, \gg\!= t_2 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2\ t_1, S)$$

(Label$_1$)
$$(\Delta, pc, N \mid \mathit{label}\ t_1\ t_2, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \mathit{label}\ t_2 : S)$$

(Label$_2$)
$$\frac{pc \sqsubseteq \ell \sqsubseteq \mu.cl \qquad t^\circ = \Delta^*(t)}{(\Delta, pc, N \mid \ell, \mathit{label}\ t : S) \rightsquigarrow_\mu (\Delta, pc, N \mid \mathit{return}\ (\mathit{Labeled}\ \ell\ t^\circ), S)}$$

(Unlabel$_1$)
$$(\Delta, pc, N \mid \mathit{unlabel}\ t, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t, \mathit{unlabel} : S)$$

(Unlabel$_2$)
$$\frac{pc \sqcup \ell \sqsubseteq \mu.cl}{(\Delta, pc, N \mid \mathit{Labeled}\ \ell\ t, \mathit{unlabel} : S) \rightsquigarrow_\mu (\Delta, pc \sqcup \ell, N \mid \mathit{return}\ t, S)}$$

Fig. 2: Syntax and semantics of sequential LIO$_{\mathrm{PAR}}$.

Semantics. The state of the monitor, written $(\Delta, pc, N \mid t, S)$, stores the state of a thread under execution and consists of a heap Δ that maps variables to terms, the thread's program counter label pc, the set N containing the identifiers of the thread's children, the term currently under reduction t and a stack of continuations S. Fig. 2 shows the interesting rules of the sequential small-step operational semantics of the security monitor. The notation $s \rightsquigarrow_{\mu} s'$ denotes a transition of the machine in state s that reduces to state s' in one step with thread parameters $\mu = (h, cl)$.[3] Since we are interested in modeling a system with *finite* resources, we parameterize the transition with the maximum heap size $h \in \mathbb{N}$. Additionally, the clearance label cl represents an upper bound over the sensitivity of the thread's floating counter label pc. Rule [APP$_1$] begins a function application. Since our calculus is call-by-need, the function argument is saved as a *thunk* (i.e., an unevaluated expression) on the heap at fresh location x and the indirection is pushed on the stack for future lookups.[4]

Note that the rule allocates memory on the heap, thus the premise $|\Delta| < h$ forbids a heap overflow, where the notation $|\Delta|$ denotes the size of the heap Δ, i.e., the number of bindings that it contains.[5] To avoid overflows, a thread can measure the size of its own heap via primitive *size* (§4.2). If t_1 evaluates to a function, e.g., $\lambda y.t$, rule [APP$_2$] starts evaluating the body, in which the bound variable y is substituted with the heap-allocated argument x, i.e., $t\ [x\ /\ y]$. When the evaluation of the function body requires the value of the argument, variable x is looked up in the heap (rule [VAR]). In the next paragraph we present the rules of the basic security primitives. The other sequential rules are available in Appendix A.

Security Primitives. A labeled value $Labeled\ \ell\ t^{\circ}$ of type $Labeled\ \tau$ consists of term t of type τ and a label ℓ, which reflects the sensitivity of the content.[6] The annotation t° denotes that term t is *closed* and does not contain any free variable, i.e., $fv(t) = \varnothing$. We restrict the syntax of labeled values with closed terms for security reasons. Intuitively, **MAC** allocates free variables inside a secret labeled values on the heap, which then leaks information to public threads with its size. For example, a public thread could distinguish between two secret values, e.g., **Labeled** $H\ x$ with heap $\Delta = [x \mapsto 42]$, and **Labeled** $H\ 0$ with heap $\Delta = \varnothing$, by measuring the size of the heap. To avoid that, labeled values are closed and the size of the heap of a thread at a certain security level, is not affected by data labeled at different security levels. A term of type $LIO\ \tau$ is a secure computation that performs side effects and returns a result of type τ. Secure computations are structured using standard monadic constructs $return\ t$,

[3] We use record notation, i.e., $\mu.h$ and $\mu.cl$, to access the components of μ.

[4] The calculus does not feature lazy evaluation. *Sharing* introduces the lazy covert channel, which has already been considered in previous work [151].

[5] To simplify reasoning, our generic memory model is basic and just counts the number of bindings in the heap. It would be possible to replicate our results with more accurate memory models, e.g., GHC's tagless G-machine (STG) [109] (the basis for GHC's runtime [88]), but that would complicate the formalism.

[6] The typing rules are standard and omitted.

which embeds term t in the monad, and *bind*, written $t_1 \gg\!= t_2$, which sequentially composes two monadic actions, the second of which takes the result of the first as an argument. Rule [$\textsc{Bind}_1$] deconstructs a computation $t_1 \gg\!= t_2$ into term t_1 to be reduced first and pushes on the stack the continuation $\gg\!= t_2$ to be invoked after term t_1. [7]

Then, the second rule [$\textsc{Bind}_2$] pops the topmost continuation placed on the stack (i.e., $\gg\!= t_2$) and evaluates it with the result of the first computation (i.e., $t_2\ t_1$), which is considered complete when it evaluates to a monadic value, i.e., to syntactic form $return\ t_1$. The runtime monitor secures the interaction between computations and labeled values. In particular, secure computations can construct and inspect labeled values exclusively with monadic primitives $label$ and $unlabel$ respectively. Rules [$\textsc{Label}_1$] and [$\textsc{Unlabel}_1$] are straightforward and follow the pattern seen in the other rules. Rule [$\textsc{Label}_2$] generates a labeled value at security level ℓ, subject to the constraint $pc \sqsubseteq \ell \sqsubseteq cl$, which prevents a computation from labeling values below the program counter label pc or above the clearance label cl.[8] The rule computes the closure of the content, i.e., closed term t°, by recursively substituting every free variable in term t with its value in the heap, written $\Delta^*(t)$. Rule [$\textsc{Unlabel}_2$] extracts the content of a labeled value and taints the program counter label with its label, i.e., it rises it to $pc \sqcup \ell$, to reflect the sensitivity of the data that is now in scope. The premise $pc \sqcup \ell \sqsubseteq cl$ ensures that the program counter label does not float over the clearance cl. Thus, the run-time monitor prevents the program counter label from floating above the clearance label (i.e., $pc \sqsubseteq cl$ always holds).

The calculus also includes concurrent primitives to allocate resources when forking threads ($fork$ and $spawn$ in §4.1), reclaim resources and measure resource usage ($kill$, $size$, and $time$ in §4.2), threads synchronization and communication ($wait$, $send$ and $receive$ in Appendix A).

4.1 Core Scheduler

In this section, we extend $\mathbf{LIO}_{\mathrm{PAR}}$ with concurrency, which enables (i) *interleaved* execution of threads on a single core and (ii) *simultaneous* execution on κ cores. To protect against attacks that exploit the automatic management of shared *finite* resource (e.g., those in §2.3), $\mathbf{LIO}_{\mathrm{PAR}}$ maintains a resource budget for each running thread and updates it as threads allocate and reclaim resources. Since κ threads execute at the same time, those changes must be coordinated in order to preserve the consistency of the resource budgets and guarantee *deterministic parallelism*. For this reason, the hierarchical runtime system is split in two components: (i) the *core scheduler*, which executes threads on a single core, ensures that they respect their resource budgets and performs security checks, and (ii) the top-level *parallel scheduler*, which synchronizes the

[7] Even though the stack size is unbounded in this model, we could account for its memory usage by explicitly allocating it on the heap, in the style of Yang et al. [162].

[8] The labels form a security lattice $(\mathscr{L}, \sqcup, \sqsubseteq)$.

$$\begin{array}{llll} \text{Thread Map} & T \in \mathit{TId} \rightharpoonup \mathit{State} & \text{Global State} & \Sigma ::= (T, B, H, \theta, \omega) \\ \text{Time Map} & B \in \mathit{TId} \rightharpoonup \mathbb{N} & \text{Core Queue} & Q ::= \langle n^b \rangle \mid \langle Q_1 \mid Q_2 \rangle \\ \text{Size Map} & H \in \mathit{TId} \rightharpoonup \mathbb{N} & \text{Event} & e ::= \epsilon \mid \mathbf{fork}(\Delta, n, t, b, h) \\ \text{Core Map} & \theta \in \mathit{TId} \rightharpoonup \mathcal{P}(\{1 \mathinner{..} \kappa\}) & & \mid \mathbf{spawn}(\Delta, n, t, K) \\ \text{Clock} & \omega \in \mathbb{N} & & \mid \mathbf{kill}(n) \mid \mathbf{send}(n, t) \end{array}$$

(Step)

$$\frac{\Sigma.T(n) = s \qquad \mu = (\Sigma.H(n), n.cl) \qquad s \rightsquigarrow_\mu s'}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s', \epsilon)}_\Sigma Q[\langle n^b \rangle]}$$

(Fork)

$$\frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N \mid b_2, \mathit{fork}\ \ell_L\ \ell_H\ h_2\ t : S) \\ pc \sqsubseteq \ell_L \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(\ell_L, \ell_H, n.k) \\ s = (\Delta, pc, \{n'\} \cup N \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \\ \Sigma.H(n) = h_1 + h_2 \qquad |\Delta| \leqslant h_1 \quad |\Delta'| \leqslant h_2 \end{array}}{Q[\langle n^{1+b_1+b_2} \rangle] \xrightarrow{(n, s, \mathbf{fork}(\Delta', n', t, b_2, h_2))}_\Sigma Q[\langle \langle n^{b_1} \rangle \mid \langle n'^{b_2} \rangle \rangle]}$$

(Spawn)

$$\frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N \mid k, \mathit{spawn}\ \ell_L\ \ell_H\ K_1\ t : S) \\ \Sigma.\theta(n) = \{k\} \cup K_1 \cup K_2 \qquad pc \sqsubseteq \ell_L \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(\ell_L, \ell_H, k) \\ s = (\Delta, pc, \{n'\} \cup N \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \end{array}}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s, \mathbf{spawn}(\Delta', n', t, K_1))}_\Sigma Q[\langle n^b \rangle]}$$

(Stuck)

$$\frac{\begin{array}{c} \Sigma.T(n) = s \qquad \mathit{MaxHeapSize}(s, \Sigma.H(n)) \vee \mathit{UnlabelStuck}(n, \Sigma.T) \vee \\ \mathit{ForkStuck}(n, \Sigma.H, \Sigma.T) \vee \mathit{SpawnStuck}(s, \theta(n)) \vee \mathit{ValueStuck}(s) \vee \\ \mathit{WaitStuck}(n, T) \vee \mathit{ReceiveStuck}(s) \vee \mathit{KillStuck}(s) \end{array}}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s, \epsilon)}_\Sigma Q[\langle n^b \rangle]}$$

(ContextSwitch)

$$\frac{s_\circ = ([\,], \bot, \varnothing \mid \mathit{return}\ (), [\,])}{Q[\langle n^0 \rangle] \xrightarrow{(\circ, s_\circ, \epsilon)}_\Sigma Q[\langle n_1^{\Sigma.B(n_1)} \rangle, \ldots, \langle n_{|Q|}^{\Sigma.B(n_{|Q|})} \rangle]}$$

Fig. 3: Concurrent $\mathbf{LIO}_{\mathrm{PAR}}$.

execution on multiple cores and reassigns resources by updating the resource budgets according to the instructions of the core schedulers. We now introduce the core scheduler and describe the top-level parallel scheduler in §4.3.

Syntax. Fig. 3 presents the core scheduler, which has access to the global state $\Sigma = (T, B, H, \theta, \omega)$, consisting of a thread pool map T, which maps a thread id to the corresponding thread's current state, the time budget map B, a memory budget map H, core capabilities map θ, and the global clock ω. Using these maps, the core scheduler ensures that thread n: (i) performs $B(n)$

uninterrupted steps until the next thread takes over, (ii) does not grow its heap above its maximum heap size $H(n)$, and (iii) has exclusive access to the *free* core capabilities $\theta(n)$. Furthermore, each thread id n records the *initial* current label when the thread was created ($n.pc$), its clearance ($n.cl$), and the core where it runs ($n.k$), so that the runtime system can enforce security. Notice that thread ids are *opaque* to threads—they cannot forge them nor access their fields.

Hierarchical Scheduling. The core scheduler performs *deterministic* and *hierarchical* scheduling—threads lower in the hierarchy are scheduled first, i.e., parent threads are scheduled before their children. The scheduler manages a core run queue Q, which is structured as a binary tree with leaves storing thread ids and residual time budgets. The notation n^b indicates that thread n can run for b more steps before the next thread runs. When a new thread is spawned, the scheduler creates a subtree with the parent thread on the left and the child on the right. The scheduler can therefore find the thread with the highest priority by following the left spine of the tree and backtracking to the right if a thread has no residual budget.[9] We write $Q[\langle n^b \rangle]$ to mean the first thread encountered via this traversal is n with budget b. As a result, given the slice $Q[\langle n^{1+b} \rangle]$, thread n is the next thread to run, and $Q[\langle n^0 \rangle]$ occurs only if *all* threads in the queue have zero residual budget. We overload this notation to represent tree updates: a rule $Q[\langle n^{1+b} \rangle] \to Q[\langle n^b \rangle]$ finds the next thread to run in queue Q and decreases its budget by one.

Semantics. Fig. 3 formally defines the transition $Q \xrightarrow{(n,s,e)}_{\Sigma} Q'$, which represents an execution step of the *core scheduler* that schedules thread n in core queue Q, executes it with global state $\Sigma = (T, B, H, \theta, \omega)$ and updates the queue to Q'. Additionally, the core scheduler informs the parallel scheduler of the final state s of the thread and requests on its behalf to update the global state by means of event message e. In rule [Step], the scheduler retrieves the next thread in the schedule, i.e., $Q[\langle n^{1+b} \rangle]$ and its state in the thread pool from the global state, i.e., $\Sigma.T(n) = s$. Then, it executes the thread for one sequential step with its memory budget and clearance, i.e., $s \rightsquigarrow_{\mu} s'$ with $\mu = (\Sigma.H(n), n.cl)$, sends the empty event ϵ to the parallel scheduler, and decrements the thread's residual budget in the final queue, i.e., $Q[\langle n^b \rangle]$. In rule [Fork], thread n creates a new thread t with initial label ℓ_{L} and clearance ℓ_{H}, such that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ and $pc \sqsubseteq \ell_{\mathrm{L}}$. The child thread runs on the same core of the parent thread, i.e., $n.k$, with fresh id n', which is then added to the set of children, i.e., $\{ n' \} \cup N$. Since parent and child threads do not share memory, the core scheduler must copy the portion of the parent's private heap reachable by the child's thread, i.e., Δ'; we do this by copying the bindings

[9] This procedure might reintroduce a timing channel that leaks the number of threads running on the core. In practice, techniques from real time schedulers could be used to protect against such timing channels. The model of **LIO**$_{\mathrm{PAR}}$ does not capture the execution time of the runtime system itself and thus this issue does not arise in the security proofs.

of the variables that are transitively reachable from t, i.e., $fv^*(t, \Delta)$, from the parent's heap Δ. The parent thread gives h_2 of its memory budget $\Sigma.H(n)$ to its child. The conditions $|\Delta| \leqslant h_1$ and $|\Delta'| \leqslant h_2$, ensure that the heaps do not overflow their new budgets. Similarly, the core scheduler splits the residual time budget of the parent into b_1 and b_2 and informs the parallel scheduler about the new thread and its resources with event $\mathbf{fork}(\Delta', n', t, b_2, h_2)$, and lastly updates the tree Q by replacing the leaf $\langle n^{1+b_1+b_2} \rangle$ with the two-leaves tree $\langle \langle n^{b_1} \rangle | \langle n'^{b_2} \rangle \rangle$, so that the child thread will be scheduled immediately after the parent has consumed its remaining budget b_1, as explained above. Rule [Spawn] is similar to [Fork], but consumes core capability resources instead of time and memory. In this case, the core scheduler checks that the parent thread owns the core where the child is scheduled and the core capabilities assigned to the child, i.e., $\theta(n) = \{k\} \cup K_1 \cup K_2$ for some set K_2, and informs the parallel scheduler with event $\mathbf{spawn}(\Delta', n', t, K_1)$. Rule [Stuck] performs busy waiting by consuming the time budget of the scheduled thread, when it is *stuck* and cannot make any progress—the premises of the rule enumerate the conditions under which this can occur (see Fig. 7 in Appendix A for details). Lastly, in rule [ContextSwitch] all the threads scheduled in the core queue have consumed their time budget, i.e., $Q[\langle n^0 \rangle]$ and the core scheduler resets their residual budget using the budget map $\Sigma.B$. In the rule, the notation $Q[\langle n_i^b \rangle]$ selects the i-th leaf, where $i \in \{1 \,..\, |Q|\}$ and $|Q|$ denotes the number of leaves of tree Q and symbol $\circ$ denotes the thread identifier of the core scheduler, which updates a dummy thread that simply spins during a context-switch or whenever the core is unused.

4.2 Resource Reclamation and Observations

The calculus presented so far enables threads to manage their time, memory and core capabilities hierarchically, but does not provide any primitive to reclaim their resources. This section rectifies this by introducing (i) a primitive to kill a thread and return its resources back to the owner and (ii) a primitive to elicit a garbage collection cycle and reclaim unused memory. Furthermore, we demonstrate that the runtime system presented in this paper is robust against timing attacks by exposing a timer API allowing threads to access a global clock.[10] Intuitively, it is secure to expose this feature because $\mathbf{LIO}_{\mathrm{PAR}}$ ensures that the time spent executing high threads is fixed in advanced, so timing measurements of low threads remain unaffected. Lastly, since memory is hierarchically partitioned, each thread can securely query the current size of its *private heap*, enabling fine-grained control over the garbage collector.

Kill. A parent thread can reclaim the resources given to its child thread n', by executing $kill\ n'$. If the child thread has itself forked or spawned other threads, they are transitively killed and their resources returned to the parent thread.

[10] An *external* attacker can take timing measurements using network communications. An attacker equipped with an *internal* clock is equally powerful but simpler to formalize [107].

$$\text{(KILL}_2\text{)}\quad \frac{\Sigma.T(n) = (\Delta, pc, \{\, n' \,\} \cup N \mid n', \mathit{kill} : S) \qquad s = (\Delta, pc, N \mid \mathit{return}\ (), S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\mathbf{kill}(n'))}_{\Sigma} Q[\langle n^{b} \rangle]}$$

$$\text{(UNLABEL}_2\text{)}\quad \frac{pc \sqcup \ell \sqsubseteq \mu.cl \qquad \forall\, n \in N \,.\, pc \sqcup \ell \sqsubseteq n.pc}{(\Delta, pc, N \mid \mathit{Labeled}\ \ell\ t, \mathit{unlabel} : S) \rightsquigarrow_{\mu} (\Delta, pc \sqcup \ell, N \mid \mathit{return}\ t, S)}$$

$$\text{(GC)}\quad \frac{R = fv^*(t, \Delta) \cup fv^*(S, \Delta) \qquad \Delta' = \{\, x \mapsto \Delta(x) \mid x \in R \,\}}{\langle \Delta, pc, N \mid \mathit{gc}\ t, S \rangle \rightsquigarrow_{\mu} \langle \Delta', pc, N \mid t, S \rangle}$$

$$\text{(APP-GC)}\quad \frac{|\Delta| \equiv \mu.h}{\langle \Delta, pc, N \mid t_1\ t_2, S \rangle \rightsquigarrow_{\mu} \langle \Delta, pc, N \mid \mathit{gc}\ (t_1\ t_2), S \rangle}$$

$$\text{(SIZE)}\quad \langle \Delta, pc, N \mid \mathit{size}, S \rangle \rightsquigarrow_{\mu} \langle \Delta, pc, N \mid \mathit{return}\ |\Delta|, S \rangle$$

$$\text{(TIME)}\quad \frac{\Sigma.T(n) = (\Delta, pc, N \mid \mathit{time}, S) \qquad s = (\Delta, pc, N \mid \mathit{return}\ \Sigma.\omega, S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\epsilon)}_{\Sigma} Q[\langle n^{b} \rangle]}$$

Fig. 4: **LIO**$_{\mathrm{PAR}}$ with resource reclamation and observation primitives.

The concurrent rule [KILL$_2$] in Fig. 4 initiates this process, which is completed by the parallel scheduler via event $\mathbf{kill}(n')$. Note that the rule applies only when the thread killed is a *direct* child of the parent thread—that is when the parent's children set has shape $\{\, n' \,\} \cup N$ for some set N. Now that threads can unrestrictedly reclaim resources by killing their children, we must revise the primitive *unlabel*, since the naive combination of *kill* and *unlabel* can result in information leakage. This will happen if a public thread forks another public thread, then reads a secret value (raising its label to secret), and based on that decides to kill the child. To close the leak, we modify the rule [UNLABEL$_2$] by adding the highlighted premise, causing the primitive *unlabel* to fail whenever the parent thread's label would float above the *initial* current label of one of its children.

Garbage Collection. Rule [GC] extends **LIO**$_{\mathrm{PAR}}$ with a *time-sensitive hierarchical* garbage collector via the primitive $gc\ \ t$. The rule elicits a garbage collection cycle which drops entries that are no longer needed from the heap, and then evaluates t. The sub-heap Δ' includes the portion of the current heap that is (transitively) *reachable* from the free variables in scope (i.e., those present in the term, $fv^*(t, \Delta)$ or on the stack $fv^*(S, \Delta)$). After collection, the

thread resumes and evaluates term t under compacted private heap Δ'.[11] In rule [App-GC], a collection is *automatically* triggered when the thread's next memory allocation would overflow the heap.

Resource Observations. All threads in the system share a global fine-grained clock ω, which is incremented by the parallel scheduler at each cycle (see below). Rule [Time] gives all threads unrestricted access to the clock via monadic primitive $time$.

4.3 Parallel Scheduler

This section extends $\mathbf{LIO}_{\mathrm{PAR}}$ with *deterministic parallelism*, which allows to execute κ threads simultaneously on as many cores. To this end, we introduce the top-level parallel scheduler, which coordinates simultaneous changes to the global state by updating the resource budgets of the threads in response core events (e.g., fork, spawn, and kill) and ticks the global clock.

Semantics. Fig. 5 formalizes the operational semantics of the parallel scheduler, which reduces a configuration $c = \langle \Sigma, \Phi \rangle$ consisting of global state Σ and core map Φ mapping each core to its run queue, to configuration c' in one step, written $c \hookrightarrow c'$, through rule [Parallel] only. The rule executes the threads scheduled on each of the κ cores, which all step at once according to the concurrent semantics presented in §4.1–4.2, with the same current global state Σ. Since the execution of each thread can change Σ *concurrently*, the top-level parallel scheduler reconciles those actions by updating Σ *sequentially* and *deterministically*.[12] First, the scheduler updates the thread pool map T and core map Φ with the final state obtained by running each thread in isolation, i.e., $T' = \Sigma.T[n_i \mapsto s_i]$ and $\Phi' = \Phi[i \mapsto Q_i]$ for $i \in \{1 \,..\, \kappa\}$. Then, it collects all concurrent events generated by the κ threads together with their thread id, sorts the events according to type, i.e., $sort\ [(n_1, e_1), ..., (n_\kappa, e_\kappa)]$, and computes the updated configuration by processing the events in sequence.[13] In particular, new threads are created first (event $\mathbf{spawn}(\cdot)$ and $\mathbf{fork}(\cdot)$), and then killed (event $\mathbf{kill}(\cdot)$)—the ordering between events of the same type is arbitrary and assumed to be fixed. Trivial events (ϵ) do not affect the configuration and thus their ordering is irrelevant. The function $\langle\!\langle es \rangle\!\rangle^c$ computes a final configuration by processing a list of events in order, accumulating configuration updates ($next(\cdot)$ updates the current configuration by one event-step): $\langle\!\langle (n, e) : es \rangle\!\rangle^c = \langle\!\langle es \rangle\!\rangle^{next(n,e,c)}$. When no more events need processing, the configuration is returned $\langle\!\langle [\,] \rangle\!\rangle^c = c$.

[11] In practice a garbage collection cycle takes time that is proportional to the size of the memory used by the thread. That does not hinder security as long as the garbage collector runs on the thread's time budget.

[12] Non-deterministic updates would make the model vulnerable to refinement attacks [90].

[13] Since the clock only needs to be incremented, we could have left it out from the configuration $c = \langle T', B, H, \theta, \Sigma.\omega + 1, \Phi' \rangle$; function $\langle\!\langle es \rangle\!\rangle^c$ does not use nor change its value.

Queue Map $\Phi \in \{1\,..\,\kappa\} \to \mathit{Queue}$ Configuration $c ::= \langle T, B, H, \theta, \omega, \Phi \rangle$

(Parallel)

$$\frac{\begin{array}{c}\forall\, i \in \{1\,..\,\kappa\}.\Phi(i) \xrightarrow{(n_i, s_i, e_i)}_{\Sigma} Q_i \\ T' = \Sigma.T[n_i \mapsto s_i] \qquad \Phi' = \Phi[i \mapsto Q_i] \\ c = \langle T', B, H, \theta, \Sigma.\omega + 1, \Phi' \rangle \qquad \langle \Sigma', \Phi'' \rangle = \langle\!\langle sort\ [(n_1, e_1), ..., (n_\kappa, e_\kappa)] \rangle\!\rangle^c\end{array}}{\langle \Sigma, \Phi \rangle \hookrightarrow \langle \Sigma', \Phi'' \rangle}$$

$$\begin{array}{l} next(\_, \epsilon, c) = c \\ next(n_1, \mathbf{fork}(\Delta, n_2, t, b, h), c) \\ \quad = \langle T', B'[n_2 \mapsto b], H'[n_2 \mapsto h], \theta', \omega, \Phi \rangle \\ \quad \mathbf{where}\ \langle T, B, H, \theta, \omega, \Phi \rangle = c \\ \qquad\qquad s = (\Delta, n_2.pc, \varnothing \mid t, [\,]) \\ \qquad\qquad T' = T[n_2 \mapsto s] \\ \qquad\qquad B' = B[n_1 \mapsto B(n_1) - b] \\ \qquad\qquad H' = H[n_1 \mapsto H(n_1) - h] \\ \qquad\qquad \theta' = \theta[n_2 \mapsto \varnothing] \\ next(n_1, \mathbf{spawn}(\Delta, n_2, t, K), c) \\ \quad = \langle T', B', H', \theta'[n_2 \mapsto K], \omega, \Phi' \rangle \\ \quad \mathbf{where}\ \langle T, B, H, \theta, \omega, \Phi \rangle = c \\ \qquad\qquad s = (\Delta, n_2.pc, \varnothing \mid t, [\,]) \\ \qquad\qquad T' = T[n_2 \mapsto s] \\ \qquad\qquad B' = B[n_2 \mapsto B_0] \\ \qquad\qquad H' = H[n_2 \mapsto H_0] \\ \qquad\qquad \theta' = \theta[n_1 \mapsto \theta(n_1) \setminus \{\, n_2.k \,\} \cup K] \\ \qquad\qquad \Phi' = \Phi[n_2.k \mapsto \langle n_2^{B_0} \rangle] \\ next(n, \mathbf{kill}(n'), \langle T, B, H, \theta, \omega, \Phi \rangle) \\ \quad \mid\ n \notin Dom(T) = \langle T, B, H, \theta, \omega, \Phi \rangle \\ \quad \mid\ n \in Dom(T) = \langle T \setminus N, B' \setminus N, H' \setminus N, \theta' \setminus N, \omega, \Phi' \rangle \\ \quad \mathbf{where}\ N = [\![\{\, n' \,\}]\!]^T \\ \qquad\qquad B' = B[n \mapsto B(n) + \textstyle\sum_{i \in N, i.k = n.k} B(i)] \\ \qquad\qquad H' = H[n \mapsto H(n) + \textstyle\sum_{i \in N, i.k = n.k} H(i)] \\ \qquad\qquad K_1 = \textstyle\bigcup_{i \in N} \theta(i) \\ \qquad\qquad K_2 = \{\, i.k \mid i \in N, i.k \neq n.k \,\} \\ \qquad\qquad \theta' = \theta[n \mapsto \theta(n) \cup K_1 \cup K_2] \\ \qquad\qquad \Phi' = \lambda k.\Phi[k \mapsto \Phi(k) \setminus N] \end{array}$$

Fig. 5: Top-level parallel scheduler.

Event Processing. Fig. 5 defines the function $next(n, e, c)$, which updates the current configuration c according to event e from thread n. The empty event ϵ is trivial and leaves the state unchanged. Event $(n_1, \mathbf{fork}(\Delta, n_2, t, b, h))$ indicates that thread n_1 forks thread t with identifier n_2, sub-heap Δ, time budget b and maximum heap size h. The scheduler deducts these resources from the parent's budgets, i.e., $B' = B[n_1 \mapsto B(n_1) - b]$ and $H' = H[n_1 \mapsto H(n_1) - h]$ and assigns them to the child, i.e., $B'[n_2 \mapsto b]$ and $H'[n_2 \mapsto h]$.[14] The new child shares the core with the parent—it has no core capabilities i.e., $\theta' = \theta[n_2 \mapsto \varnothing]$—and so the core map is left unchanged. Lastly, the scheduler adds the child to the thread pool and initializes its state, i.e., $T[n_2 \mapsto (\Delta, n_2.\ell_{\mathrm{L}}, \varnothing \mid t, [\,])]$. The scheduler handles event $(n_1, \mathbf{spawn}(\Delta, n_2, t, K))$ similarly. The new thread t gets scheduled on core $n_2.k$, i.e., $\Phi[n_2.k \mapsto \langle n_2^{B_0} \rangle]$, where the thread takes all the time and memory resources of the core, i.e., $B[n_2 \mapsto B_0]$ and $H[n_2 \mapsto H_0]$, and extra core capabilities K, i.e., $\theta'[n_2 \mapsto K]$. For simplicity, we assume that all cores execute B_0 steps per-cycle and feature a memory of size H_0. Event $(n, \mathbf{kill}(n'))$ informs the scheduler that thread n wishes to kill thread n'. The scheduler leaves the global state unchanged if the parent thread has already been killed by the time this event is handled, i.e., when the guard $n \notin Dom(T)$ is true—the resources of the child n' will have been reclaimed by another ancestor. Otherwise, the scheduler collects the identifiers of the descendants of n' that are *alive* ($N = [\![\{\, n' \,\}]\!]^T$)—they must be killed (and reclaimed) *transitively*. The set N is computed recursively by $[\![N]\!]^T$, using the thread pool T, i.e., $[\![\varnothing]\!]^T = \varnothing$, $[\![\{\, n \,\}]\!]^T = \{\, n \,\} \cup [\![T(n).N]\!]^T$ and $[\![N_1 \cup N_2]\!]^T = [\![N_1]\!]^T \cup [\![N_2]\!]^T$. The scheduler then increases the time and memory budget of the parent with the sum of the budget of all its descendants scheduled on the *same* core, i.e., $\sum_{i \in N, i.k = n.k} B(i)$ (resp. $\sum_{i \in N, i.k = n.k} H(i)$)—descendants running on other cores do not share those resources. The scheduler reassigns to the parent thread their core capabilities, which are split between capabilities explicitly assigned but not in use, i.e., $\bigcup_{i \in N} \theta(i)$ and core capabilities assigned and in use by running threads, i.e., $\{\, i.k \mid i \in N, i.k \neq n.k \,\}$. Lastly, the scheduler removes the killed threads from each core, written $\Phi(i) \setminus N$, by pruning the leaves containing killed threads and reassigning their leftover time budget to their parent, see Appendix A.2 for details.

5 Security Guarantees

In this section we show that $\mathbf{LIO}_{\mathrm{PAR}}$ satisfies a strong security condition that ensures timing-agreement of threads and rules out timing covert channels. In §5.1, we describe our proof technique based on *term erasure*, which has been used to verify security guarantees of functional programming languages [76], IFC libraries [30, 55, 141, 145, 153]), and an IFC runtime system [151]. In §5.2, we formally prove security, i.e., *timing-sensitive noninterference*, a strong form of noninterference [48], inspired by Volpano and Smith [158]—to our knowledge, it is considered here for the first time in the context of parallel runtime systems.

[14] Notice that $|\Delta| < h$ by rule [Fork].

Works that do not address external timing channels [151, 154] normally prove *progress-sensitive* noninterference, wherein the number of execution steps of a program may differ in two runs based on a secret. This condition is insufficient in the parallel setting: both public and secret threads may step simultaneously on different cores and any difference in the number of execution steps would introduce *external* and *internal* timing attacks. Similar to previous works on secure multi-threaded systems [84, 129], we establish a *strong* low-bisimulation property of the parallel scheduler, which guarantees that configurations that are indistinguishable to the attacker remain such and execute in lock-step. Theorem 1 and Corollary 1 use this property to ensure that any two related parallel programs execute in exactly the same number of steps.

5.1 Erasure Function

The term erasure technique relies on an *erasure function*, written $\varepsilon_L(\cdot)$, which rewrites secret data above the attacker's level L to special term $\bullet$, in all the syntactic categories: values, terms, heaps, stacks, global states and configurations.[15] Once the erasure function is defined, the core of the proof technique consists of proving an essential *commutativity* relationship between the erasure function and reduction steps: given a step $c \hookrightarrow c'$, there must exist a reduction that *simulates* the original reduction between the erased configurations, i.e., $\varepsilon_L(c) \hookrightarrow \varepsilon_L(c')$. Intuitively, if the configuration c leaked secret data while stepping to c', that data would be classified as public in c' and thus would remain in $\varepsilon_L(c')$— but such secret data would be erased by $\varepsilon_L(c)$ and the property would not hold. The erasure function leaves ground values, e.g., $()$, unchanged and on most terms it acts homomorphically, e.g., $\varepsilon_L(t_1\ t_2) = \varepsilon_L(t_1)\ \varepsilon_L(t_2)$. The interesting cases are for labeled values, thread configurations, and resource maps. The erasure function removes the content of secret labeled values, i.e., $\varepsilon_L(\mathit{Labeled}\ H\ t^\circ) = \mathit{Labeled}\ H\ \bullet$, and erases the content recursively otherwise, i.e., $\varepsilon_L(\mathit{Labeled}\ L\ t^\circ) = \mathit{Labeled}\ L\ \varepsilon_L(t)^\circ$. The state of a thread is erased per-component, homomorphically if the program counter label is public, i.e., $\varepsilon_L(\Delta, L, N, \mid t, S) = (\varepsilon_L(\Delta), L, N \mid \varepsilon_L(t), \varepsilon_L(S))$, and in full otherwise, i.e., $\varepsilon_L(\Delta, H, N, \mid t, S) = (\bullet, \bullet, \bullet \mid \bullet, \bullet)$. We give the full definition in Appendix B.

Resource Erasure. Resources must also be appropriately erased in order to satisfy the simulation property, as $\mathbf{LIO}_{\mathrm{PAR}}$ manages resources explicitly. The erasure function should *preserve* information about the resources (e.g., time, memory, and core capabilities) of *public threads*, since the attacker can explicitly assign resources (e.g., with fork and swap) and measure them (e.g., with size). But what about the resources of secret threads? One might think that such information is secret and thus it should be erased—intuitively, a thread might decide to assign, say, half of its time budget to its secret child depending on secret information. However, public threads can also assign (public) resources to a secret thread when forking: even though these resources currently belong

[15] For ease of exposition, we use the two-point lattices $\{L, H\}$, where $H \not\sqsubseteq L$ is the only disallowed flow. Neither our proofs nor our model rely on this particular lattice.

to the secret child, they are *temporary*—the public parent might reclaim them later. Thus, we cannot associate the sensitivity of the resources of a thread with its program counter label when resources are managed *hierarchically*, as in **LIO**$_{\mathrm{PAR}}$. Instead, we associate the security level of the resources of a secret thread with the sensitivity of its parent: the resources of a secret thread are *public* information whenever the program counter label of the parent is public and *secret* information otherwise. Furthermore, since resource reclamation is transitive, the erasure function cannot discard secret resources, but must rather redistribute them to the hierarchically closest set of public resources, as when *kill*ing them.

Time Budget. First, we project the identifiers of *public* threads from the thread pool T: $Dom_L(T) = \{n_L \mid n \in Dom(T) \wedge T(n).pc \equiv L\}$, where notation n_L indicates that the program counter label of thread n is public. Then, the set $P = \bigcup_{n \in Dom_L(T)} \{n\} \cup T(n).N$ contains the identifiers of all the public threads and their immediate children.[16] The resources of threads $n \in P$ are public information. However, the program counter label of a thread $n \in P$ is not necessarily public, as explained previously. Hence P can be disjointly partitioned by program counter label: $P = P_L \cup P_H$, where $P_L = \{n_L \mid n \in P\}$ and $P_H = \{n_H \mid n \in P\}$. Erasure of the budget map then proceeds on this partition, leaving the budget of the public threads untouched, and summing the budget of their secret children threads to the budgets of their descendants, which are instead omitted. In symbols, $\varepsilon_L(B) = B_L \cup B_H$, where $B_L = \{n_L \mapsto B(n_L) \mid n_L \in P_L\}$ and $B_H = \{n_H \mapsto B(n_H) + \sum_{i \in [\![\{n_H\}]\!]^T} B(i) \mid n_H \in P_H\}$.

Queue Erasure. The erasure of core queues follows the same intuition, preserving public and secret threads $n \in P$ and trimming all other secret threads $n_H \notin P$. Since queues annotate thread ids with their residual time budgets, the erasure function must reassign the budgets of all *secret* threads $n'_H \notin P$ to their closest ancestor $n \in P$ on the same core. The ancestor $n \in P$ could be either (i) another *secret* thread on the same core, i.e., $n_H \in P$, or, (ii) the spinning thread of that core, $\circ \in P$ if there is no other thread $n \in P$ on that core—the difference between these two cases lies on whether the original thread n' was *forked* or *spawned* on that core. More formally, if the queue contains no thread $n \in P$, then the function replaces the queue altogether with the spinning thread and returns the residual budgets of the threads to it, i.e., $\varepsilon_L(Q) = \langle \circ^B \rangle$ if $n_i \notin P$ and $B = \sum b_i$, for each leaf $Q[\langle n_i^{b_i} \rangle]$ where $i \in \{1 \,..\, |Q|\}$. Otherwise, the core contains at least a thread $n_H \in P$ and the erasure function returns the residual time budget of its secret descendants, i.e., $\varepsilon_L(Q) = Q{\downarrow_L}$ by combining the effects of the following mutually recursive functions:

[16] The id of the spinning thread on each free core is also public, i.e., $\circ_k \in P$ for $k \in \{1 \,..\, \kappa\}$.

$$\langle n^b \rangle \downarrow_L = \langle n^b \rangle \qquad \langle n_{1H}^{b_1} \rangle \curlyvee \langle n_{2H}^{b_2} \rangle = \langle n_{1H}^{b_1+b_2} \rangle$$
$$\langle Q_1, Q_2 \rangle \downarrow_L = (Q_1 \downarrow_L) \curlyvee (Q_2 \downarrow_L) \qquad Q_1 \curlyvee Q_2 = \langle Q_1, Q_2 \rangle$$

The interesting case is $\langle n_{1H}^{b_1} \rangle \curlyvee \langle n_{2H}^{b_2} \rangle$, which reassigns the budget of the child (the right leaf $\langle n_{2H}^{b_2} \rangle$) to the parent (the left leaf $\langle n_{1H}^{b_1} \rangle$), by rewriting the subtree into $\langle n_{1H}^{b_1+b_2} \rangle$.

5.2 Timing-Sensitive Non-interference

The proof of timing-sensitive noninterference relies on two fundamental properties, i.e., *determinancy* and *simulation* of parallel reductions. Determinancy requires that the reduction relation is deterministic.

Proposition 1 (Determinism). *If $c_1 \hookrightarrow c_2$ and $c_1 \hookrightarrow c_3$ then $c_2 \equiv c_3$.*

The equivalence in the statement denotes alpha-equivalence, i.e., up to the choice of variable names. We now show that the parallel scheduler preserves L-equivalence of parallel configurations.

Definition 1 (L-equivalence). *Two configurations c_1 and c_2 are indistinguishable from an attacker at security level L, written $c_1 \approx_L c_2$, if and only if $\varepsilon_L(c_1) \equiv \varepsilon_L(c_2)$.*

Proposition 2 (Parallel Simulation). *Given a parallel reduction step $c \hookrightarrow c'$, then $\varepsilon_L(c) \hookrightarrow \varepsilon_L(c')$.*

By combining *determinism* (Proposition 1) and *parallel simulation* (Proposition 2), we prove *progress-insensitive noninterference*, which assumes progress of both configurations.

Proposition 3 (Progress-Insensitive Non-interference). *If $c_1 \hookrightarrow c_1'$, $c_2 \hookrightarrow c_2'$ and $c_1 \approx_L c_2$, then $c_1' \approx_L c_2'$.*

In order to lift this result to be timing-sensitive, we first prove *time sensitive progress*. Intuitively, if a valid [17] configuration steps then any low equivalent parallel configuration also steps.

Proposition 4 (Time-Sensitive Progress). *Given a valid configuration c_1 and a parallel reduction step $c_1 \hookrightarrow c_1'$ and $c_1 \approx_L c_2$, then there exists c_2', such that $c_2 \hookrightarrow c_2'$.*

Using progress-insensitive noninterference, i.e., Proposition 3 and time-sensitive progress, i.e., Proposition 4 in combination, we obtain a *strong* L-bisimulation property between configurations and prove *timing-sensitive noninterference.*

[17] A configuration is valid if satisfies several basic properties, e.g., it does not contain special term $\bullet$. See Appendix B for details

Theorem 1 (Timing-Sensitive Non-interference). *For all valid configurations c_1 and c_2, if $c_1 \hookrightarrow c_1'$ and $c_1 \approx_L c_2$, then there exists a configuration c_2', such that $c_2 \hookrightarrow c_2'$ and $c_1' \approx_L c_2'$.*

The following corollary instantiates the timing-sensitive noninterference security theorem for a given **LIO**$_{\text{PAR}}$ parallel program, that explicitly rules out leaks via timing channels. In the following, the notation $\hookrightarrow_u$, denotes u parallel reduction steps, as usual.

Corollary 1. *Given a well-typed **LIO**$_{\text{PAR}}$ program t of type $Labeled\ \tau_1 \rightarrow$ **LIO** τ_2 and two secret closed terms $t_1^\circ, t_2^\circ :: \tau_1$, let $i \in \{1,2\}, j \in \{1..\kappa\}$, n_L be a thread identifier such that $n.k = 1$ and $n.cl = H$, and*

- $s_i = ([\,], L, \varnothing, \mid t\ (Labeled\ H\ t_i^\circ), [\,])$
- $T_i = [n_L \mapsto s_i, \circ_j \mapsto s_\circ]$
- $B = [n_L \mapsto B_0, \circ_j \mapsto 0]$
- $H = [n_L \mapsto H_0, \circ_j \mapsto H_0]$
- $\theta = [n_L \mapsto \{2..\kappa\}, \circ_j \mapsto \varnothing]$
- $\Phi_i = [1 \mapsto \langle s_i \rangle, 2 \mapsto \langle \circ_2 \rangle, ..., \kappa \mapsto \langle \circ_\kappa \rangle]$
- $c_i = (T_i, B, H, \theta, 0, \Phi_i)$

If $c_1 \hookrightarrow_u c_1'$, then there exists configuration c_2', such that $c_2 \hookrightarrow_u c_2'$ and $c_1' \approx_L c_2'$.

To conclude, we show that the *timing-sensitive* security guarantees of **LIO**$_{\text{PAR}}$ extend to concurrent *single-core* programs by instantiating Corollary 1 with $\kappa = 1$.

6 Limitations

Implementation. Implementing **LIO**$_{\text{PAR}}$ is a serious undertaking that requires a major redesign of GHC's runtime system. Conventional runtime systems freely share resources among threads to boost performance and guarantee fairness. For instance, in GHC, threads share heap objects to save memory space and execution time (when evaluating expressions). In contrast, **LIO**$_{\text{PAR}}$ strictly partitions resources to enforce security—threads at different security labels cannot share heap objects. As a result, the GHC memory allocator must be adapted to isolate threads' private heap, so that allocation and collection can occur independently and in parallel. Similarly, the GHC "fair" round robin scheduler must be heavily modified to keep track of and manage threads' time budget, to preemptively perform a context switch when their time slice is up.

Programming model. Since resource management is explicit, building applications atop **LIO**$_{\text{PAR}}$ introduces new challenges—the programmer must explicitly choose resource bounds for each thread. If done poorly, threads can spend excessive amounts of time sitting idle when given too much CPU time, or garbage collecting when not given enough heap space. The problem of tuning resource allocation parameters is not unique to **LIO**$_{\text{PAR}}$—Yang and Mazières' [162] propose to use GHC profiling mechanisms to determine heap size while the real-time garbage collector by Henriksson [54] required the

programmer to specify the worst case execution time, period, and worst-case allocation of each high-priority thread. Das and Hoffmann [36] demonstrate a more automatic approach—they apply machine learning techniques to statically determine upper bounds on execution time and heap usage of OCaml programs. Similar techniques could be applied to $\mathbf{LIO}_{\mathrm{PAR}}$ in order to determine the most efficient resource partitions. We further remark that this challenge is not unique to real-time systems or $\mathbf{LIO}_{\mathrm{PAR}}$; choosing privacy parameters in differential privacy shares many similarities [60,74]. Even though $\mathbf{LIO}_{\mathrm{PAR}}$ programming model might seem overly restrictive, we consider it appropriate for certain classes of applications (e.g., web applications and certain embedded systems). To further simplify programming with $\mathbf{LIO}_{\mathrm{PAR}}$, we intend to introduce privileges (and thus declassification) similar to LIO [46, 145] or COWL [146]. Floating-label systems such as LIO and $\mathbf{LIO}_{\mathrm{PAR}}$ often suffer from *label creep* issues, wherein the current label gets tainted to a point where the computation cannot perform any useful side-effects [144]. Similar to concurrent LIO [145], $\mathbf{LIO}_{\mathrm{PAR}}$ relies on primitive *fork* to address label creep[18], but, at the cost of a restricted floating-label mechanisms, $\mathbf{LIO}_{\mathrm{PAR}}$ provides also parallel execution, garbage collection, and APIs for heap statistics, elapsed time, and kill.

7 Related work

There is substantial work on language-level IFC systems [43, 51, 59, 80, 101, 122, 123, 141, 144, 163, 164]. Our work builds on these efforts in several ways. Firstly, $\mathbf{LIO}_{\mathrm{PAR}}$ extends the concurrent LIO IFC system [141] with parallelism—to our knowledge, this is the first *dynamic* IFC system to support parallelism and address the internalization of external timing channels. Previous static IFC systems implicitly allow for parallelism, e.g., Muller and Chong's [95], several works on IFC π-calculi [57, 58, 66], and Rafnsson et al. [118] recent foundations for composable timing-sensitive interactive systems. These efforts, howerver, do not model runtime system resource management. Volpano and Smith [158] enforce a timing agreement condition, similar to ours, but for a static concurrent IFC system. Mantel et al. [83] and Li et al. [77] prove noninterference for static, concurrent systems, using rely-guarantee reasoning.

Unlike most of these previous efforts, our hierarchical runtime system also eliminates classes of resource-based external timing channels, such as memory exhaustion and garbage collection. Pedersen and Askarov [107], however, were the first to identify automatic memory management to be a source of covert channels for IFC systems and demonstrate the feasibility of attacks against both V8 and the JVM. They propose a sequential static IFC language with labeled-partitioned memory and a label-aware timing-sensitive garbage collec-

[18] Sequential LIO addresses label creep through primitive $\mathbf{toLabeled}(\cdot)$, which executes a computation (that may raise the current label) in a separate context and restores the current label upon its termination. **MAC** does not feature $\mathbf{toLabeled}(\cdot)$, because the primitive opens the termination covert-channel and thus is not timing-sensitive [141].

tor, which is vulnerable to *external timing* attacks and satisfies only *termination-insensitive* non-interference.

Previous work on language-based systems—namely [81, 162]—identify memory retention and memory exhaustion as a source of denial-of-service (DOS) attacks. Memory retention and exhaustion can also be used as covert channels. In addressing those covert channels, $\mathbf{LIO}_{\mathrm{PAR}}$ also addresses the DOS attacks outlined by these efforts. Indeed, our work generalizes Yang and Mazières' [162] region-based allocation framework with region-based garbage collection and hierarchical scheduling.

Our $\mathbf{LIO}_{\mathrm{PAR}}$ design also borrows ideas from the secure operating system community. Our explicit hierarchical memory management is conceptually similar to HiStar's container abstraction [166]. In HiStar, containers—subject to quotas, i.e., space limits—are used to hierarchically allocate and deallocate objects. $\mathbf{LIO}_{\mathrm{PAR}}$ adopts this idea at the language-level and automates the allocation and reclamation. Moreover, we hierarchically partition CPU-time; Zeldovich et al. [166], however, did observe that their container abstraction can be repurposed to enforce CPU quotas.

Deterland [161] splits time into ticks to address internal timing channels and mitigate external timing ones. Deterland builds on Determinator [10], an OS that executes parallel applications deterministically and efficiently. $\mathbf{LIO}_{\mathrm{PAR}}$ adopts many ideas from these systems—both the deterministic parallelism and ticks (semantic steps)—to the language-level. Deterministic parallelism at the language-level has also been explored previous to this work [70, 71, 87], but, different from these efforts, $\mathbf{LIO}_{\mathrm{PAR}}$ also hierarchically manages resources to eliminate classes of external timing channels.

Fabric [79, 80] and DStar [167] are distributed IFC systems. Though we believe that our techniques would scale beyond multi-core systems (e.g., to data centers), $\mathbf{LIO}_{\mathrm{PAR}}$ will likely not easily scale to large distributed systems like Fabric and DStar. Different from Fabric and DStar, however, $\mathbf{LIO}_{\mathrm{PAR}}$ addresses both internal and external timing channels that result from running code in parallel.

Our hierarchical resource management approach is not unique—other countermeasures to external timing channels have been studied. Hu [61], for example, mitigates both timing channels in the VAX/VMM system [78] using "fuzzy time"—an idea recently adopted to browsers [68]. Askarov et al.'s [5] mitigate external timing channels using predicative black-box mitigation, which delays events and thus bound information leakage. Rather than using noise as in the fuzzy time technique, however, they predict the schedule of future events. Some of these approaches have also been adopted at the language-level [107, 141, 168]. We find these techniques largely orthogonal: they can be used alongside our techniques to mitigate timing channels we do not eliminate.

Real-time systems—when developed with garbage collected languages [6, 13, 23, 54]—face similar challenges as this work. Blelloch and Cheng [23] describe a real-time garbage collector (RTGC) for multi-core programs with *prov-*

able resource bounds—**LIO**$_{\mathrm{PAR}}$ *enforces* resource bounds instead. A more recent RTGC created by Auerbach et al. [6] describes a technique to "tax" threads into contributing to garbage collection as they utilize more resources. Henriksson [54] describes a RTGC capable of enforcing hard and soft deadlines, once given upper bounds on space and time resources used by threads. Most similarly to **LIO**$_{\mathrm{PAR}}$, Pizlo et al. [114] implement a hierarchical RTGC algorithm that independently collects partitioned heaps.

8 Conclusion

Language-based IFC systems built atop off-the-shelf runtime systems are vulnerable to resource-based external-timing attacks. When these systems are extended with thread parallelism the attacks become yet more vicious—they can be carried out internally. We presented **LIO**$_{\mathrm{PAR}}$, the design of the first dynamic IFC hierarchical runtime system that support deterministic parallelism and eliminates both resource-based internal- and external-timing covert channels. To our knowledge, **LIO**$_{\mathrm{PAR}}$ is the first parallel system to satisfy progress- and time-sensitive non-interference.

References

1. Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.
2. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.
3. Joshua Auerbach, David F Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 245–254. ACM, 2008.
4. Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.
5. Henry G Baker Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.
6. Guy E Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *ACM SIGPLAN Notices*, volume 34, pages 104–117. ACM, 1999.
7. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.
8. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proceedings of the 18th Nordic Conference on Secure IT Systems - Volume 8208*, NordSec 2013, pages 116–122, New York, NY, USA, 2013. Springer-Verlag New York, Inc.
9. Ankush Das and Jan Hoffmann. ML for ML: Learning cost semantics by experiment. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construc-*

tion and Analysis of Systems, pages 190–207, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

10. Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium*, pages 531–548, 2016.
11. Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27, 2016.
12. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.
13. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. *Journal of Computer Security*, 25, 2017.
14. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy*, pages 75–75, April 1984.
15. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing*. ACM, 2014.
16. Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, 1998.
17. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 11–31, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
18. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
19. Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure Information Flow as Typed Process Behaviour. In *European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.
20. Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 81–92, New York, NY, USA, 2002. ACM.
21. C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
22. Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C. Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 398–410, Washington, DC, USA, 2014. IEEE Computer Society.
23. Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4):233–254, 1992.
24. Richard A Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems (TOCS)*, 1(3):256–277, 1983.

25. Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4):291–347, December 2005.
26. David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, pages 463–480, 2016.
27. Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.
28. Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. *ACM SIGPLAN Notices*, 49(6):2–14, 2014.
29. Jaewoo Lee and Chris Clifton. How much is enough? choosing ϵ for differential privacy. In *Proceedings of the 14th International Conference on Information Security*, ISC'11, pages 325–340, Berlin, Heidelberg, 2011. Springer-Verlag.
30. Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, April 2010.
31. Ximeng Li, Heiko Mantel, and Markus Tasch. Taming message-passing communication in compositional reasoning about confidentiality. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, pages 45–66, 2017.
32. S. Lipner, T. Jaeger, and M. E. Zurko. Lessons from vax/svs for high-assurance vm systems. *IEEE Security Privacy*, 10(6):26–35, Nov 2012.
33. Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.
34. Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.
35. Jed Liu and Andrew C Myers. Defining and enforcing referential security. In *International Conference on Principles of Security and Trust*, pages 199–219. Springer, 2014.
36. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 218–232, June 2011.
37. Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Comput. Secur.*, 11(4):615–676, July 2003.
38. Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *ACM SIGPLAN Notices*, 46(12):71–82, 2012.
39. Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.
40. D. McCullough. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy(SP)*, volume 00, page 161, April 1987.
41. Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, pages 57–74, New York, NY, USA, October 2012. ACM Press.
42. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.

43. Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *USENIX Security Symposium*, pages 1119–1136, 2016.
44. Stephen C North and John H Reppy. Concurrent garbage collection on stock hardware. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–133. Springer, 1987.
45. James Lee Parker. *LMonad: Information flow control for haskell web applications*. PhD thesis, University of Maryland, College Park, 2014.
46. Mathias V. Pedersen and Aslan Askarov. From trash to treasure: Timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 693–709, 2017.
47. Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.
48. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
49. Filip Pizlo, Antony L. Hosking, and Jan Vitek. Hierarchical real-time garbage collection. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 123–133, New York, NY, USA, 2007. ACM.
50. Willard Rafnsson, Limin Jia, and Lujo Bauer. Timing-sensitive noninterference through composition. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 3–25, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
51. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2009.
52. Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 280–288, New York, NY, USA, 2015. ACM.
53. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, CSFW '00, pages 200–, Washington, DC, USA, 2000. IEEE Computer Society.
54. Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, May 1997.
55. Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.
56. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Nordic Conference on Security IT Systems (NordSec)*. Springer, October 2011.
57. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27, 2017.
58. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.
59. Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with

COWL. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014.

60. Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 187–202, Washington, DC, USA, 2007. IEEE Computer Society.
61. Marco Vassena, Joachim Breitner, and Alejandro Russo. Securing concurrent lazy programs against information leakage. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 37–52, 2017.
62. Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 538–557, 2016.
63. Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 15–28, New York, NY, USA, 2016. ACM.
64. Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. Mac a verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming*, 2017.
65. Pepe Vila and Boris Kopf. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 849–864, Vancouver, BC, 2017. USENIX Association.
66. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.
67. Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in deterland. In *Conference on Timely Results in Operating Systems, Monterey, CS, US*, 2015.
68. Edward Z. Yang and David Mazières. Dynamic space limits for haskell. *SIGPLAN Not.*, 49(6):588–598, June 2014.
69. Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *ACM SIGPLAN Notices*, volume 51, pages 631–647. ACM, 2016.
70. Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.
71. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.
72. Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.
73. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 563–574, New York, NY, USA, 2011. ACM.

Appendix

A Full Calculus

Context Rules. In Fig. 6, for completeness, we report the remaining context rules of sequential **LIO**$_{\mathrm{PAR}}$—they rules simply evaluate their arguments by pushing (popping) the appropriate continuation on the stack.

Stuck Conditions. Fig. 7 formally defines the conditions used in rule [Stuck], which identify a thread as *stuck*. When adding garbage collection, i.e., rule [App-GC], we also remove the condition $MaxHeapStuck(s)$ from rule [Stuck]—such condition triggers an automatic garbage collection cycle that reduces via rule [GC].

Reachable Variables and Term Closure. Fig. 8 presents two operations that compute and substitute free variables. Fig. 8a formally defines the set of transitively reachable free variables for all syntactic categories, i.e., stacks, continuations and message queues. The function simply computes the free variables by induction on the structure of terms, stacks, continuation and message queues recursively. Notice that labeled values are closed and thus the function returns the empty set for them, i.e., $fv^*(Labeled\ \ell\ t^\circ, \Delta) = \varnothing$. Fig. 8b defines a closure function, i.e., $\Delta^*_B(t)$, which, given an open term t with bound variables B and a heap Δ, computes the corresponding closed term t° by recursively substituting free variables to close the term. When the function traverses a lambda expression, i.e., $\lambda x.t$, it recurs in the body and adds variable x to the set of bound variables, i.e., $\Delta^*_{B\cup\{x\}}(t)$. When the function finds a free variable x, it looks it up in the heap and repeats the process by closing the corresponding thunk, i.e., $\Delta^*(\Delta(x))$. When omitted, the set of bound variables is empty, i.e., $\Delta^*(t) = \Delta^*_\varnothing(t)$. Notice that the recursion is well-funded because our heap does not contain cyclic definitions (the syntax of the calculus does not feature recursive let bindings and therefore every term can be closed).

(IF$_1$)
$(\Delta, pc, N \mid \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \textbf{then } t_2 \textbf{ else } t_3 : S)$

(IF$_2$)
$(\Delta, pc, N \mid \mathit{True}, \textbf{then } t_2 \textbf{ else } t_3 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2, S)$

(IF$_3$)
$(\Delta, pc, N \mid \mathit{False}, \textbf{then } t_2 \textbf{ else } t_3 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_3, S)$

(FORK$_1$)
$(\Delta, pc, N \mid \mathit{fork}\ t_1\ t_2\ t_3\ t_4\ t_5, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \mathit{fork}\ t_2\ t_3\ t_4\ t_5 : S)$

(FORK$_2$)
$(\Delta, pc, N \mid \ell, \mathit{fork}\ t_2\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2, \mathit{fork}\ \ell\ t_3\ t_4\ t_5 : S)$

(FORK$_3$)
$(\Delta, pc, N \mid \ell_2, \mathit{fork}\ \ell_1\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_3, \mathit{fork}\ \ell_1\ \ell_2\ t_4\ t_5 : S)$

(FORK$_4$)
$(\Delta, pc, N \mid h, \mathit{fork}\ \ell_1\ \ell_2\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_4, \mathit{fork}\ \ell_1\ \ell_2\ h\ t_5 : S)$

(SPAWN$_1$)
$(\Delta, pc, N \mid \mathit{spawn}\ t_1\ t_2\ t_3\ t_4\ t_5, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \mathit{spawn}\ t_2\ t_3\ t_4\ t_5 : S)$

(SPAWN$_2$)
$(\Delta, pc, N \mid \ell, \mathit{spawn}\ t_2\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2, \mathit{spawn}\ \ell\ t_3\ t_4\ t_5 : S)$

(SPAWN$_3$)
$(\Delta, pc, N \mid \ell_2, \mathit{spawn}\ \ell_1\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_3, \mathit{spawn}\ \ell_1\ \ell_2\ t_4\ t_5 : S)$

(SPAWN$_4$)
$(\Delta, pc, N \mid K, \mathit{spawn}\ \ell_1\ \ell_2\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_4, \mathit{spawn}\ \ell_1\ \ell_2\ K\ t_5 : S)$

(WAIT$_1$)
$(\Delta, pc, N \mid \mathit{wait}\ t, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t, \mathit{wait} : S)$

(KILL$_1$)
$(\Delta, pc, N \mid \mathit{kill}\ t, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t, \mathit{kill} : S)$

Fig. 6: Context rules of sequential **LIO**$_{\mathrm{PAR}}$.

(ForkStuck)

$$\frac{T(n) = (\_, pc, \_ \mid \ell_2, \mathit{fork}\ \ell_1\ t : S) \quad \ell_1\ \not\sqsubseteq\ \ell_2 \vee pc\ \not\sqsubseteq\ \ell_1 \vee H(n) = h_1 + h_2 \quad |\Delta| > h_1 \vee |\Delta'| > h_2}{\mathit{ForkStuck}(n, H, T)}$$

(UnlabelStuck)

$$\frac{T(n) = (\_, pc, N \mid \mathit{Labeled}\ \ell\ t, \mathit{unlabel} : S) \quad pc\ \sqcup\ \ell\ \not\sqsubseteq\ n.cl \vee \exists\, n\ \in\ N\,.\, pc\ \sqcup\ \ell\ \not\sqsubseteq\ n.pc}{\mathit{UnlabelStuck}(n, T)}$$

(WaitStuck)

$$\frac{T(n) = (\_, pc, \_ \mid n', \mathit{wait} : S) \quad n.cl\ \not\sqsubseteq\ pc\ \vee\ \mathit{eval}(T)(n') = n' \not\mapsto (\_, \_, \_ \mid v, [\,])}{\mathit{WaitStuck}(n, T)}$$

(ValueStuck)

$$\frac{s = (\_, \_, \_ \mid v, [\,])}{\mathit{ValueStuck}(s)}$$

(MaxHeapSize)

$$\frac{s = (\Delta, \_, \_ \mid t_1\ t_2, \_) \quad |\Delta| = h}{\mathit{MaxHeapSize}(s, h)}$$

(SpawnStuck)

$$\frac{s = (\Delta, pc, N \mid k, \mathit{spawn}_{K_1}\ \ell_1\ \ell_2\ t : S) \quad pc\ \not\sqsubseteq\ \ell_1\ \vee\ \ell_1\ \not\sqsubseteq\ \ell_2\ \vee\ k \cup K_1 \not\subseteq K}{\mathit{SpawnStuck}(s, K)}$$

(KillStuck)

$$\frac{s = (\Delta, pc, N \mid n, \mathit{kill} : S) \quad n\ \notin\ N}{\mathit{KillStuck}(s)}$$

(ReceiveStuck)

$$\frac{s = (\Delta, pc, N, [\,] \mid \mathit{receive}, S)}{\mathit{ReceiveStuck}(s)}$$

Fig. 7: Stuck conditions.

$$
\begin{aligned}
fv^*(x, \Delta) &= \{\, x \,\} \cup fv^*(\Delta(x), \Delta) \\
fv^*(\lambda x.t, \Delta) &= fv^*(t, \Delta) \setminus \{\, x \,\} \\
fv^*(t_1\ t_2, \Delta) &= fv^*(t_1, \Delta) \cup fv^*(t_2, \Delta) \\
fv^*(\mathit{Labeled}\ \ell\ t^\circ, \Delta) &= \varnothing
\end{aligned}
$$

$$
\begin{aligned}
fv^*([\,], \Delta) &= \varnothing \\
fv^*(C : S, \Delta) &= fv^*(C, \Delta) \cup fv^*(S, \Delta)
\end{aligned}
$$

$$
\begin{aligned}
fv^*(x, \Delta) &= \{\, x \,\} \cup fv^*(\Delta(x), \Delta) \\
fv^*(\gg\!= t, \Delta) &= fv^*(t, \Delta) \\
fv^*(\mathit{label}\ t, \Delta) &= fv^*(t, \Delta)
\end{aligned}
$$

$$
\begin{aligned}
fv^*([\,], \Delta) &= \varnothing \\
fv^*(t \lhd ts, \Delta) &= fv^*(t, \Delta) \cup fv^*(ts, \Delta)
\end{aligned}
$$

(a) Transitively-reachable free variables (excerpt).

$$
\begin{aligned}
\Delta^*_B(x) &= \begin{cases} x & \text{if } x \in B \\ \Delta^*(\Delta(x)) & \text{if } x \notin B \end{cases} \\
\Delta^*_B(t_1\ t_2) &= \Delta^*_B(t_1)\ \Delta^*_B(t_2) \\
\Delta^*_B(\lambda x.t) &= \Delta^*_{B \cup \{\, x \,\}}(t)
\end{aligned}
$$

(b) Term closure (excerpt).

Fig. 8: Free variables manipulation.

A.1 Thread Synchronization and Communication

$\mathbf{LIO}_{\mathrm{PAR}}$ features primitive $wait\ n'$, which allows a thread to wait on the result of some thread n', see Rule [WAIT] in Fig. 9. If the thread has terminated, i.e., its term is a value and the stack of continuations is empty, its value is returned to the waiting thread. The condition $n'.cl \sqsubseteq pc$ ensures that the waited-upon thread has an appropriate clearance (indicating that its label must be low enough to observe its result) for security reasons. In all other cases, the [STUCK] rule applies via the [WAITSTUCK] condition.

Furthermore, $\mathbf{LIO}_{\mathrm{PAR}}$ features also thread communication primitives, which enable a programming model akin to the actor model [56]. In particular, primitive $send\ n\ t$ enables *asynchronous* best-effort delivery of message t to thread n—there are no guarantees that the message will be delivered. Note that rule [SEND_2] simply generates an event that instructs the top-level parallel scheduler to deliver a message, which then might get dropped for security reasons or otherwise. Conversely, rule [RECEIVE] executes *synchronous* primitive $receive$, which inspects the thread's messages queue ts and extracts the next message—if the queue is empty, the thread gets stuck.[19] The rule generates event $\mathbf{send}(n_2, t)$, so that the parallel scheduler delivers message t to thread n_2, by means of function $next(\cdot)$. The scheduler drops the message if the receiver is dead, i.e., $n_2 \notin Dom(T)$, or if the sender is dead, i.e., $n_2 \notin Dom(T)$. If the receiver has sufficient memory budget, i.e., $|\Delta_2'| < H(n_2)$ and is at least as sensitive as the sender, i.e., $pc_1 \sqsubseteq pc_2$, then the scheduler delivers and enqueues the message in the message queue, i.e., $ts_2 \triangleright t$, or drops it otherwise.[20]

[19] The receiver thread will also get stuck if the type of the message does not match the expected type. We are not concerned with the type-safety of primitive $receive$, which could be recovered with dynamic typing.

[20] This feature makes threads vulnerable to Denial of Service (DOS) attacks, in which the attacker floods a thread with messages until it runs out of memory. We could restore security by adding message integrity or by pre-allocating a memory budget for queue messages.

$$\begin{array}{lll} \text{Message Queue} & ts & ::= [\,] \mid t \triangleleft ts \\ \text{State} & s & ::= (\Delta, pc, N, ts \mid t, S) \end{array}$$

(WAIT)

$$\frac{T(n) = (\Delta, pc, N, ts \mid n', wait : S) \quad n'.cl \sqsubseteq pc \quad T(n') = (\_, \_, \_, \_ \mid return\ v, [\,]) \quad s = (\Delta, pc, N, ts \mid v, S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\epsilon)}_{\Sigma} Q[\langle n^{b} \rangle]}$$

(SEND$_1$)

$$(\Delta, pc, N, ts \mid send\ t_1\ t_2, S) \rightsquigarrow (\Delta, pc, N, ts, \mid t_1, send\ t_2 : S)$$

(SEND$_2$)

$$\frac{n \mapsto (\Delta, pc, N, ts \mid n', send\ t : S) \in \Sigma.T \quad s = (\Delta, pc, N, ts \mid return\ (), S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\mathbf{send}(n',t))}_{\Sigma} Q[\langle n^{b} \rangle]}$$

(RECEIVE)

$$\frac{n \mapsto (\Delta, pc, N, t \triangleleft ts \mid receive, S) \in \Sigma.T \quad s = (\Delta, pc, N, ts \mid return\ t, S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\epsilon)}_{\Sigma} Q[\langle n^{b} \rangle]}$$

$$\begin{array}{l} next(n_1, \mathbf{send}(n_2, t), \langle T, B, H, \theta, \Phi \rangle) \\ \quad \mid n_1 \notin Dom(T) \vee n_2 \notin Dom(T) = \langle T, B, H, \theta, \Phi \rangle \\ \quad \mid |\Delta_2'| < H(n_2) \wedge pc_1 \sqsubseteq pc_2 = \langle T', B, H, \theta, \Phi \rangle \\ \quad \mid otherwise = \langle T, B, H, \theta, \Phi \rangle \\ \quad \textbf{where}\ (\Delta_1, pc_1, \_, \_ \mid \_, \_) = T(n_1) \\ \qquad\quad (\Delta_2, pc_2, N_2, ts_2 \mid t_2, S_2) = T(n_2) \\ \qquad\quad \Delta_2' = \Delta_2 \cup \{ x \mapsto \Delta_1(x) \mid x \in fv^*(t, \Delta_1) \} \\ \qquad\quad T' = T[n_2 \mapsto (\Delta_2', pc_2, N_2, ts_2 \triangleright t \mid t_2, S_2)] \end{array}$$

Fig. 9: Thread synchronization and communication primitives.

$$Q \setminus \varnothing = Q$$
$$Q \setminus (\{\, n \,\} \cup N) = \mathbf{delete}(Q, n) \setminus N$$

(a) $Q \setminus N$ removes nodes N from the tree Q.

$$\mathbf{delete}(\langle n^{\,b} \rangle, n) = \langle \circ^{0} \rangle$$

$$\mathbf{delete}(\langle \langle n^{\,b} \rangle |\ Q \rangle, n') = \begin{cases} \mathbf{add}(Q, b) & \text{if } n \equiv n' \\ \langle \langle n^{\,b} \rangle |\ \mathbf{delete}(Q, n') \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{delete}(\langle Q\ | \langle n^{\,b} \rangle \rangle, n') = \begin{cases} \mathbf{add}(Q, b) & \text{if } n \equiv n' \\ \langle \mathbf{delete}(Q, n')\ | \langle n^{\,b} \rangle \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{delete}(\langle Q_1 \mid Q_2 \rangle, n) = \begin{cases} \langle \mathbf{delete}(Q_1, n) \mid Q_2 \rangle & \text{if } n \in Q_1 \\ \langle Q_1, \mathbf{delete}(Q_2, n) \rangle & \text{if } n \in Q_2 \end{cases}$$

(b) $\mathbf{delete}(Q, n)$ removes node $n \in Q$.

$$\mathbf{add}(\langle n^{\,b_1} \rangle, b_2) = \langle n^{\,b_1 + b_2} \rangle$$

$$\mathbf{add}(\langle Q_1 \mid Q_2 \rangle, b) = \langle \mathbf{add}(Q_1, b) \mid Q_2 \rangle$$

(c) $\mathbf{add}(Q, b)$ adds b to the budget of the leftmost thread.

Fig. 10: Functions that trim and readjust the budget in a queue tree.

A.2 Queue Pruning

When threads get killed, the parallel scheduler removes them from the core queue where they are scheduled. We write $Q \setminus N$, for the queue obtained by removing threads N from queue Q, by repeated calls to function $\mathbf{delete}(\cdot)$, see Fig. 10. Function $\mathbf{delete}(Q, n)$ locates thread n in the queue and trims its leaf—if the thread is alone in the queue, it is replaced with the dummy busy-waiting thread id, i.e., $\circ$, and the core becomes free. Note that the current budget of a killed thread is not discarded, but it is reassigned to its closest relative (either to the parent, the immediate child or the spinning thread), by means of function $\mathbf{add}(\cdot)$, for security reasons. Otherwise, a high thread could influence the schedule on its own core by forking a thread and giving up, say, half of its budget, and then killing it immediately afterwards. If the parent does not regain the child's current time budget, then nothing would run for the time the child was originally given, which could lead to information leakage, when executed in parallel with other cores.

Term	t	$::=$	$\cdots \mid \bullet \mid \mathit{label}_L\ t_1\ t_2 \mid \mathit{unlabel}_L\ t \mid \mathit{fork}_L\ t_1\ t_2\ t_3\ t_4\ t_5 \mid \mathit{spawn}_L\ t_1\ t_2\ t_3\ t_4\ t_5$
Cont.	C	$::=$	$\cdots \mid \mathit{label}_L\ t \mid \mathit{unlabel}_L \mid \mathit{fork}_L\ t_1\ t_2\ t_3\ t_4 \mid \mathit{spawn}_L\ t_1\ t_2\ t_3\ t_4$
Stack	S	$::=$	$\cdots \mid \bullet$
Queue	ts	$::=$	$\cdots \mid \bullet$
Event	e	$::=$	$\cdots \mid \mathbf{fork}_L(n, b, h) \mid \mathbf{spawn}_L(\Delta, n, K)$

Fig. 11: Calculus with erased terms.

B Security Proofs

B.1 Two-Steps Erasure

During execution, a public thread might create secret data, often by elevating low data. For example, primitive $\mathit{label}\ \ell\ t$ labels a piece of data t with label ℓ in rule [LABEL$_2$] from Figure 2. Depending on the sensitivity of the label, the erasure function needs to rewrite t to either $\bullet$, if $\ell \equiv H$ or $\varepsilon_L(t)$ if $\ell \equiv L$, in order to respect the simulation property for rule [LABEL$_2$]. Unfortunately, the label is a runtime value, thus it might not be known when we apply the erasure function, e.g., in rule [LABEL$_1$]. *Two-steps erasure* is a technique that extends term erasure and simplify reasoning about such operations, especially when the decision of erasing data depends on the context or on runtime values [153]. In particular, this technique ensures that those operations commute under the erasure function by rewriting problematic primitives with new ad-hoc constructs, that erase terms at runtime, when sufficient information is available. Figure 11 extends the calculus with erasure-aware primitives. Fig. 12 and 16 define the erasure function for all the syntactic categories of **LIO**$_{\mathrm{PAR}}$, which rewrites the problematic terms with the those extra primitives, which are reduced according to the rules in Fig. 13 and 15. Equipped with those extra primitives, we use term erasure with two-steps erasure to prove security of **LIO**$_{\mathrm{PAR}}$ in the next section.

Example. The erasure function rewrites *label* to new construct label_L, which firstly evaluates the label via rule [LABEL$_{L1}$] and then erase the second argument as needed, when the value of the label is known in rules [LABEL$_{L2}$] and [LABEL$_{L3}$].

$$\varepsilon_L(()) = () \qquad \varepsilon_L(\ell) = \ell \qquad \varepsilon_L(\lambda x.e) = \lambda x.\varepsilon_L(e)$$

$$\varepsilon_L(\mathit{return}\ e) = \mathit{return}\ \varepsilon_L(e) \qquad \varepsilon_L(n) = n \qquad \varepsilon_L(k) = k$$

$$\varepsilon_L(\mathit{Labeled}\ \ell\ t^\circ) = \begin{cases} \mathit{Labeled}\ \ell\ \bullet & \text{if } \ell \not\sqsubseteq L \\ \mathit{Labeled}\ \ell\ \varepsilon_L(t)^\circ & \text{otherwise} \end{cases}$$

(a) Values.

$$\varepsilon_L(\Delta) = \{x \mapsto \varepsilon_L(\Delta(x)) \mid x \in \mathit{Dom}(\Delta)\} \qquad \varepsilon_L(x) = x$$

$$\varepsilon_L(t_1\ t_2) = \varepsilon_L(t_1)\ \varepsilon_L(t_2) \qquad \varepsilon_L(t_1 \gg\!= t_2) = \varepsilon_L(t_1) \gg\!= \varepsilon_L(t_2)$$

$$\varepsilon_L(\mathit{label}\ t_1\ t_2) = \mathit{label}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2) \qquad \varepsilon_L(\mathit{unlabel}\ t) = \mathit{unlabel}_L\ \varepsilon_L(t)$$

$$\varepsilon_L(\mathit{fork}\ t_1\ t_2\ t_3\ t_4\ t_5) = \mathit{fork}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)\ \varepsilon_L(t_5)$$

$$\varepsilon_L(\mathit{spawn}\ t_1\ t_2\ t_3\ t_4\ t_5) = \mathit{spawn}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)\ \varepsilon_L(t_5)$$

$$\varepsilon_L(\mathit{wait}\ t) = \mathit{wait}\ \varepsilon_L(t)$$

(b) Heaps and terms.

$$\varepsilon_L([\,]) = [\,] \qquad \varepsilon_L(C : S) = \varepsilon_L(C) : \varepsilon_L(S) \qquad \varepsilon_L(x) = x$$

$$\varepsilon_L(\gg\!= t) = \gg\!= \varepsilon_L(t) \qquad \varepsilon_L(\mathit{label}\ t) = \mathit{label}_L\ \varepsilon_L(t)$$

$$\varepsilon_L(\mathit{unlabel}) = \mathit{unlabel}_L$$

$$\varepsilon_L(\mathit{fork}\ t_1\ t_2\ t_3\ t_4) = \mathit{fork}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)$$

$$\varepsilon_L(\mathit{spawn}\ t_1\ t_2\ t_3\ t_4) = \mathit{spawn}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)$$

$$\varepsilon_L(\mathit{wait}) = \mathit{wait}$$

(c) Stacks and continuations.

Fig. 12: Erasure for sequential $\mathbf{LIO}_{\mathrm{PAR}}$.

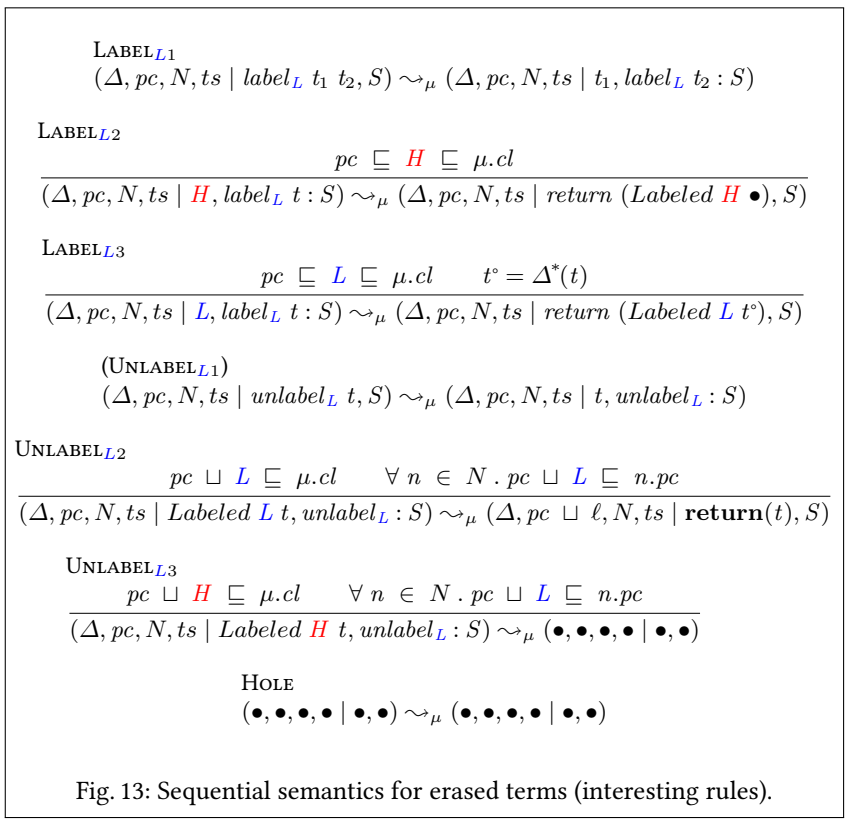


Fig. 13: Sequential semantics for erased terms (interesting rules).

$$\varepsilon_L(n_L, s, e) = (n_L, \varepsilon_L(s), \varepsilon_L(e)) \qquad \varepsilon_L(n_H, s, e) = (n_H, \varepsilon_L(s), \epsilon) \qquad \varepsilon_L(\epsilon) = \epsilon$$

$$\varepsilon_L(\mathbf{kill}(n)) = \mathbf{kill}(n) \qquad \varepsilon_L(\mathbf{send}(n, t)) = \mathbf{send}(n, \varepsilon_L(t))$$

$$\varepsilon_L(\mathbf{spawn}(\Delta, n, t, K)) = \begin{cases} \mathbf{spawn}(\varepsilon_L(\Delta), n, \varepsilon_L(t), K) & \text{if } n.pc \sqsubseteq L \\ \mathbf{spawn}_L(\varepsilon_L(\Delta), n, K) & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\mathbf{fork}(\Delta, n, t, b, h)) = \begin{cases} \mathbf{fork}(\varepsilon_L(\Delta), n, \varepsilon_L(t), b, h) & \text{if } n.pc \sqsubseteq L \\ \mathbf{fork}_L(b, h) & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\Delta, pc, N, ts \mid t, S) = \begin{cases} (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet) & \text{if } pc \not\sqsubseteq L \\ (\varepsilon_L(\Delta), pc, N, \varepsilon_L(ts) \mid \varepsilon_L(t), \varepsilon_L(S)) & \text{otherwise} \end{cases}$$

Fig. 14: Erasure for concurrent $\mathbf{LIO}_{\mathrm{PAR}}$.

(Fork$_{L5}$)

$$\frac{\begin{array}{c}\Sigma.T(n) = (\Delta, pc, N, ts \mid b_2, fork_{\,L}\ L\ \ell_{\mathrm{H}}\ h_2\ t : S) \qquad pc \sqsubseteq L \\ n' \leftarrow \mathrm{fresh}^{TId}(L, \ell_{\mathrm{H}}, n.k) \qquad s = (\Delta, pc, \{n'\} \cup N, ts \mid return\ n', S) \\ \Delta' = \{x \mapsto \Delta(x) \mid x \in fv^*(t, \Delta)\} \\ \Sigma.H(n) = h_1 + h_2 \qquad |\Delta| \leqslant h_1 \qquad |\Delta'| \leqslant h_2\end{array}}{Q[\langle n^{1+b_1+b_2}\rangle] \xrightarrow{(n,s,\mathbf{fork}(\Delta', n', t, b_2, h_2))}_{\Sigma} Q[\langle\langle n^{b_1}\rangle|\langle n'^{b_2}\rangle\rangle]}$$

(Fork$_{L6}$)

$$\frac{\begin{array}{c}\Sigma.T(n) = (\Delta, pc, N, ts \mid b_2\ fork_{\,L}\ H\ \ell_{\mathrm{H}}\ h_2\ t : S) \\ pc \sqsubseteq H \qquad n' \leftarrow \mathrm{fresh}^{TId}(H, \ell_{\mathrm{H}}, n.k) \\ s = (\Delta, pc, \{n'\} \cup N, \mid return\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in fv^*(t, \Delta)\} \\ \Sigma.H(n) = h_1 + h_2 \qquad |\Delta| \leqslant h_1 \quad |\Delta'| \leqslant h_2\end{array}}{Q[\langle n^{1+b_1+b_2}\rangle] \xrightarrow{(n,s,\mathbf{fork}_L(n', b_2, h_2))}_{\Sigma} Q[\langle\langle n^{b_1}\rangle|\langle n'^{b_2}\rangle\rangle]}$$

(Spawn$_{L4}$)

$$\frac{\begin{array}{c}\Sigma.T(n) = (\Delta, pc, N, ts \mid k, spawn_{\,L}\ L\ \ell_{\mathrm{H}}\ K_1\ t : S) \\ \Sigma.\theta(n) = \{k\} \cup K_1 \cup K_2 \qquad pc \sqsubseteq L \qquad n' \leftarrow \mathrm{fresh}^{TId}(L, \ell_{\mathrm{H}}, k) \\ s = (\Delta, pc, \{n'\} \cup N \mid return\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in fv^*(t, \Delta)\}\end{array}}{Q[\langle n^{1+b}\rangle] \xrightarrow{(n,s,\mathbf{spawn}(\Delta', n', t, K_1))}_{\Sigma} Q[\langle n^{b}\rangle]}$$

(Spawn$_{L5}$)

$$\frac{\begin{array}{c}\Sigma.T(n) = (\Delta, pc, N, ts \mid k\ spawn_{\,L}\ H\ \ell_{\mathrm{H}}\ K_1\ t : S) \\ \Sigma.\theta(n) = \{k\} \cup K_1 \cup K_2 \qquad pc \sqsubseteq H \qquad n' \leftarrow \mathrm{fresh}^{TId}(H, \ell_{\mathrm{H}}, k) \\ s = (\Delta, pc, \{n'\} \cup N \mid return\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in fv^*(t, \Delta)\}\end{array}}{Q[\langle n^{1+b}\rangle] \xrightarrow{(n,s,\mathbf{spawn}_L(\Delta', n', K_1))}_{\Sigma} Q[\langle n^{b}\rangle]}$$

Fig. 15: Concurrent semantics of erased terms.

$$Dom_L(T) = \{\, n_L \mid n \in Dom(T) \wedge T(n).pc \equiv L \,\}$$

$$P = \bigcup_{n \,\in\, Dom_L(T)} \{\, n \,\} \cup T(n).N$$

$$\varepsilon_L(T) = \{\, n \mapsto \varepsilon_L(T(n)) \mid n \in P \,\} \qquad \varepsilon_L(\Phi) = \lambda k.\varepsilon_L(\Phi(k))$$

$$\varepsilon_L(\langle T, B, H, \theta, \Phi, \omega \rangle) = \langle \varepsilon_L(T), \varepsilon_L(B), \varepsilon_L(H), \varepsilon_L(\theta), \varepsilon_L(\Phi), \omega \rangle$$

(a) Thread maps, core maps and parallel configuration.

$$P_L = \{\, n_L \mid n \in P \,\} \qquad P_H = \{\, n_H \mid n \in P \,\}$$

$$\begin{aligned}
&\varepsilon_L(B) = B_L \cup B_H \\
&\quad \textbf{where } B_L = \{\, n_L \mapsto B(n_L) \mid n_L \in P_L \,\} \\
&\qquad\quad\; B_H = \{\, n_H \mapsto B(n_H) + \textstyle\sum_{i \,\in\, [\![\{ n_H \}]\!]^T} B(i) \mid n_H \in P_H \,\} \\
&\varepsilon_L(H) = H_L \cup H_H \\
&\quad \textbf{where } H_L = \{\, n_L \mapsto H(n_L) \mid n_L \in P_L \,\} \\
&\qquad\quad\; H_H = \{\, n_H \mapsto \textstyle\sum_{i \,\in\, [\![\{ n_H \}]\!]^T, n_H.k = i.k} H(i) \mid n_H \in P_H \,\} \\
&\varepsilon_L(\theta) = \theta_L \cup \theta_H \\
&\quad \textbf{where } \theta_L = \{\, n_L \mapsto \theta(n_L) \mid n_L \in P_L \,\} \\
&\qquad\quad\; \theta_H = \{\, n_H \mapsto \theta(n_H) \cup K_1 \cup K_2 \mid n_H \in P_H \,\} \\
&\qquad\quad\; K_1 = \textstyle\bigcup_{i \,\in\, [\![\{ n_H \}]\!]^T} \theta(i) \\
&\qquad\quad\; K_2 = \{\, i.k \mid i \in [\![\{ n_H \}]\!]^T, i.k \neq n_H.k \,\}
\end{aligned}$$

(b) Time, memory and core budget maps.

$$\begin{aligned}
&next(n_{1L}, \mathbf{spawn}_L(\Delta, n_{2H}, K), \langle T, B, H, \theta, \omega, \Phi \rangle) = \langle T', B', H', \theta', \omega, \Phi' \rangle \\
&\quad \textbf{where } T' = T[n_{2H} \mapsto (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)] \\
&\qquad\quad\; B' = B[n_{2H} \mapsto B_0] \\
&\qquad\quad\; H' = H[n_{2H} \mapsto H_0] \\
&\qquad\quad\; \theta' = \theta[n_{1L} \mapsto \theta(n_{1L}) \setminus \{\, n_{2H}.k \,\} \cup K][n_{2H} \mapsto K] \\
&\qquad\quad\; \Phi' = \Phi[n_{2H}.k \mapsto \langle n_{2H}^{B_0} \rangle] \\
&next(n_{1L}, \mathbf{fork}_L(n_{2H}, b, h), \langle T, B, H, \theta, \omega, \Phi \rangle) = \langle T', B', H', \theta', \omega, \Phi \rangle \\
&\quad \textbf{where } T' = T[n_{2H} \mapsto (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)] \\
&\qquad\quad\; B' = B[n_{1L} \mapsto B(n_{1L}) - b][n_{2H} \mapsto b] \\
&\qquad\quad\; H' = H[n_{1L} \mapsto H(n_{1L}) - h][n_{2H} \mapsto h] \\
&\qquad\quad\; \theta' = \theta[n_{2H} \mapsto \varnothing]
\end{aligned}$$

(c) Processing of erased events.

Fig. 16: Erasure for parallel $\mathbf{LIO}_{\mathrm{PAR}}$.

$$\textsc{Next}_1\ \langle n^b\rangle[\langle n^b\rangle] \qquad \textsc{Next}_2\ \frac{Q_1[\langle n_1^b\rangle] \quad b>0}{\langle Q_1 \mid Q_2\rangle[\langle n_1^b\rangle]} \qquad \textsc{Next}_3\ \frac{Q_1[\langle n_1^0\rangle] \quad Q_2[\langle n_2^b\rangle]}{\langle Q_1 \mid Q_2\rangle[\langle n_2^b\rangle]}$$

(a) Hierarchical Scheduling.

$$\varepsilon_L(Q) = \begin{cases} \langle \circ^0\rangle & \text{if } Q \cap P = \varnothing \\ Q{\downarrow_L} & \text{otherwise} \end{cases}$$

$$\langle n^b\rangle{\downarrow_L} = \langle n^b\rangle \qquad \langle n_{1H}^{b_1}\rangle \curlyvee \langle n_{2H}^{b_2}\rangle = \langle n_{1H}^{b_1+b_2}\rangle$$
$$\langle Q_1, Q_2\rangle{\downarrow_L} = (Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L}) \qquad Q_1 \curlyvee Q_2 = \langle Q_1, Q_2\rangle$$

(b) Core erasure.

Fig. 17: Core queues.

B.2 Lemmas

We now prove a number of lemmas auxiliary to our security proofs. Fig. 17a formalizes the hierarchical scheduling policy and Fig. 17b repeats the erasure function for core queues, where $Q \cap P = \varnothing$ abbreviates the predicate $\forall\, i \in \{1\,..\,|Q|\}.Q[\langle n_i^{b_i}\rangle] \wedge n_i \notin P$, i.e., the queue contains only secret threads, whose resources are not observable.

Lemma 1. *If $Q[\langle n^0\rangle]$, then one of the following holds:*

1. *If $n \in P$, then $\varepsilon_L(Q)[\langle n^0\rangle]$*
2. *If $n \notin P$, then there exists $n' \in P$ such that $\varepsilon_L(Q)[\langle n'^0\rangle]$*

Proof. First, $Q[\langle n^0\rangle]$ implies that all threads in queue Q have budget 0. Then, since $\varepsilon_L(Q)$ simply redistribute the residual budgets of the secret threads on the queue, any thread scheduled by $\varepsilon_L(Q)$ will have budget 0 intuitively. Formally, we have two cases depending on whether $n \in P$ or not.

1. By induction on the scheduling policy. Case [$\textsc{Next}_1$] is trivial, case [$\textsc{Next}_2$] is impossible and case [$\textsc{Next}_3$] follows by induction. If $n \equiv \circ$ the lemma is trivial, i.e., $\varepsilon_L(\langle \circ^0\rangle) = \langle \circ^0\rangle{\downarrow_L} = \langle \circ^0\rangle$.
2. If $Q \cap P = \varnothing$, then $\varepsilon_L(Q) = \langle \circ^0\rangle$. If $Q \cap P \neq \varnothing$, we perform induction. Case [$\textsc{Next}_1$] and [$\textsc{Next}_2$] are absurd, in case [$\textsc{Next}_3$], we inspect the result of $(Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L})$. If that is a leaf, e.g., $\langle n_H^0\rangle$, then $n_H \in P_H$ (erasure reassigns the budgets of the threads in Q_1 and Q_2 to the closest ancestor in $Q \cap P$) and the lemma follows by rule [$\textsc{Next}_1$]. Otherwise, we inductively apply Lemma 1 to $Q_1[\langle n_1^0\rangle]$ and Lemma 1.2 to $Q_2[\langle n^0\rangle]$.

Lemma 2. *If* $Q[\langle n_L^b \rangle]$, *then* $\varepsilon_L(Q)[\langle n_L^b \rangle]$.

Proof. The core scheduler runs *public* thread n_L, thus $n_L \in P$ and $\varepsilon_L(Q) = Q{\downarrow_L}$. Then the proof follows by induction on the scheduling policy $Q[\langle n_L^b \rangle]$, observing in case [Next$_2$] and [Next$_3$] that $\langle Q_1, Q_2 \rangle {\downarrow_L} = \langle Q_1 {\downarrow_L}, Q_2 {\downarrow_L} \rangle$, because either Q_1 or Q_2 contains public thread n_L and using Lemma 1 in case [Next$_3$].

Lemma 3. Proof. *If* $Q[\langle n_H^{1+b} \rangle]$ *and* $n_H \in P$, *then there exists* b', *such that* $b \leqslant b'$, $\varepsilon_L(Q)[\langle n_H^{1+b'} \rangle]$.

Since $n_H \in P$, then $Q \cap P \neq \varnothing$ and $\varepsilon_L(Q) = Q{\downarrow_L}$ and we proceed by induction. Case [Next$_1$] is trivial. In case [Next$_2$], we perform case analysis on the result of $(Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L})$, if that is a branch, i.e., $\langle Q_1{\downarrow_L}, Q_2{\downarrow_L} \rangle$, then the lemma follows by induction on $Q_1[\langle n_H^{1+b} \rangle]$, otherwise, $Q_1{\downarrow_L} = \langle n_H^{1+b} \rangle$ and $Q_2{\downarrow_L} = \langle {n'}_H^{b'} \rangle$, for some other secret thread n', with leftover budget b', which then gets collapsed into the budget of its ancestor n, i.e., $\langle n_H^{1+b} \rangle \curlyvee \langle {n'}_H^{b'} \rangle = \langle n_H^{1+b+b'} \rangle$. The lemma then follows by applying rule [Next$_2$] to rule [Next$_1$], i.e., scheduling $\langle n_H^{1+b+b'} \rangle$. The same line of reasoning applies to case [Next$_3$], where the public ancestor, say ${n'}_L \in P$, of secret thread $n_H \in P$ is in the left branch, i.e., $Q_1[\langle {n'}_L^0 \rangle]$, thus $(Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L})$ equals to $\langle Q_1{\downarrow_L}, Q_2{\downarrow_L} \rangle$. Then, the lemma follows by applying Lemma 1 to $Q_1[\langle {n'}_L^0 \rangle]$ and induction on $(Q{\downarrow_L})[\langle n_H^{1+n} \rangle]$.

Lemma 4. *If* $Q[\langle n_H^{1+b} \rangle]$, $n_H \notin P$ *and* $Q \cap P \neq \varnothing$, *then there exists* ${n'}_H \in Q \cap P$ *and* $b' \geqslant b$, *such that* $\varepsilon_L(Q)[\langle {n'}_H^{1+b'} \rangle]$.

Proof. Intuitively, we need to find in core Q, the closest ancestor ${n'}_H \in Q \cap P$ of secret thread n_H — there exists one threads form a hierarchy and $Q \cap P \neq \varnothing$ — and show that the ancestor gets scheduled in the erased queue and that erasure function redistributes the residual budget to it. We do that by induction on $Q[\langle n_H^{1+b} \rangle]$. Case [Next$_1$] contradicts the hypothesis $Q \cap P \neq \varnothing$, hence the lemma is vacuously true. Case [Next$_2$] follows by induction in the left subqueue, i.e., $Q_1[\langle n_H^{1+b} \rangle]$. Intuitively, the ancestor of n_H can only be in the left subtree, thus $Q_1 \cap P \neq \varnothing$ and $\langle Q_1, Q_2 \rangle {\downarrow_L} = \langle Q_1{\downarrow_L}, Q_2{\downarrow_L} \rangle$. [21] In case [Next$_3$], we inspect the result of $(Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L})$. If that is a leaf, by Lemma 1 there exists a *secret* thread n_{1H} with zero residual budget, such that $Q_1{\downarrow_L} = \langle n_{1H}^0 \rangle$ and $Q_2{\downarrow_L} = \langle n_{2H}^{1+b_2} \rangle$ for some other thread $n_{2H} \notin Q \cap P$ with left-over budget $b_2 \geqslant b$ (since core erasure does *not* discard left-over budgets). [22] Then, the erased queue is $\langle n_{1H}^{1+b_1+b_2} \rangle$, and the lemma follows by rule [Next$_1$]. If $(Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L})$ is a branch, then either we apply Lemma 1 and induction, if

[21] The public parent thread cannot be in the right branch, i.e., Q_2, because parents are moved to the left branch when forking.

[22] Either $n_{2H} \equiv n_H$, or it is the oldest ancestor of n_H on the core.

$Q_2 \cap P \neq \varnothing$, or derive a contradiction, otherwise. Intuitively, if $Q_2 \cap P = \varnothing$, then $Q_1 \cap P \neq \varnothing$ and either (i) all the threads in the left branch are secret, i.e., $Q_1 \subseteq P_H$, which contradicts the fact that $(Q_1 \downarrow_L) \curlyvee (Q_2 \downarrow_L)$ is a branch, or (ii) there exists a public thread ${n'}_L \in Q_1$, and we can show that $n_H \in P_H$, contradicting the second hypothesis of the lemma.

Lemma 5. *Properties of the erasure function:*

1. $|\varepsilon_L(\Delta)| \equiv |\Delta|$
2. $fv^*(\varepsilon_L(\Delta), \varepsilon_L(t)) \equiv fv^*(\Delta, t)$
3. $\varepsilon_L(t_1\ [x \mathbin{/} t_2]) = \varepsilon_L(t_1)$

Proof. By straightforward induction on the arguments. The second lemma relies on secret labeled values containing *closed* term. These values have no free variables, i.e., $fv^*(\Delta, Labeled\ H\ t^\circ)) = \varnothing$, and thus the lemma holds, i.e., $fv^*(\varepsilon_L(\Delta), \varepsilon_L(Labeled\ H\ t^\circ)) = fv^*(\varepsilon_L(\Delta), Labeled\ H\ \bullet) = \varnothing$. If labeled values contained open terms, then the lemma would *not* hold, e.g., $fv^*(\Delta, Labeled\ H\ x) = \{x\} \not\equiv \varnothing = fv^*(\varepsilon_L(\Delta), Labeled\ H\ \bullet)$.

Lemma 6. *For all time budget maps B, memory budget maps H, core capabilities maps θ, core queues Q, thread pool T and public threads $n \in P$, let $N = \{n\}$, then the following hold:*

- $\sum_{i \in [\![N]\!]^T, i.k = n.k} B(i) = \sum_{j \in [\![N]\!]^{\varepsilon_L(T)}, j.k = n.k} \varepsilon_L(B)(j)$
- $\sum_{i \in [\![N]\!]^T, i.k = n.k} H(i) = \sum_{j \in [\![N]\!]^{\varepsilon_L(T)}, j.k = n.k} \varepsilon_L(H)(j)$
- $\bigcup_{i \in [\![N]\!]^T} \theta(i) = \bigcup_{i \in [\![N]\!]^{\varepsilon_L(T)}} \varepsilon_L(\theta)(i)$
- $\{i.k \mid i \in [\![N]\!]^T, i.k \neq n.k\} = \{i.k \mid i \in [\![N]\!]^{\varepsilon_L(T)}, i.k \neq n.k\}$.
- $\varepsilon_L(Q \setminus [\![N]\!]^T) = \varepsilon_L(Q) \setminus [\![N]\!]^{\varepsilon_L(T)}$.

Proof. This lemma relates the observable parts of the budget maps and the erasure function, and ensures that budget erasure returns *all* the secret resources, i.e., the resources allocated by the secret descendants of thread $n \in P$, regardless of their number. Intuitively, the left-hand side of the equation involves an arbitrary number of those secret threads and the right-hand none, because the erasure function removes them from the thread pool map. More precisely, we partition the descendants of thread $n \in P$ in two groups, based on whether the attacker can observe their resources, formally: $[\![N]\!]^T = X \cup Y$, such that $X \subseteq P$ and and $Y \not\subseteq P$. We observe that the erased thread map only contains only the threads with visible resources, i.e., $[\![N]\!]^{\varepsilon_L(T)} = X$, since $Dom(\varepsilon_L(T)) = P$, that is erasure removes the secret threads contained in Y. Then, we further partition set X in two sets depending on the security level of the threads, i.e., $X = X_L \cup X_H$, where $X_L \subseteq P_L$ and $X_H \subseteq P_H$. First, we observe that the erasure function leaves the budgets of public threads *unchanged*, i.e., $\sum_{(i_L \in X_L)} B(i_L) = \sum_{(i_L \in X_L)} B_L(i_L)$,

where $\varepsilon_L(B) = B_L \cup B_H$ from Figure 16b. Then, we only need to show that $\sum_{(i_H \in X_H \cup Y)} B(i_H) = \sum_{(i_H \in X_H)} B_H(i_H)$. The equality follows by two observations. Firstly, both sides sum the budget of the secret threads in X_H, i.e., $B(i_H)$ for $i_H \in X_H$. Secondly, the remaining secret threads $i_H \in Y \not\subseteq P$ are descendants of some secret thread $n_H \in X_H \subseteq P$ ($X \cup Y = [\![N]\!]^T$), and therefore their budget is accounted for on the right-hand side in the summation $\sum_{i \in [\![\{n_H\}]\!]^T} B(i)$. The same line of reasoning applies for memory, core capabilities budgets and core queues.

The next lemma ensures that processing any event e generated by a *secret* thread changes only the secret parts of the global state. L-equivalence of configurations (and all the other syntactic categories) is defined as the kernel of the erasure function, i.e., $c \approx_L c'$ iff $\varepsilon_L(c) \equiv \varepsilon_L(c')$, see Definition 1.

Lemma 7. *If* $next(n_H, e, c) = c'$, *then* $c \approx_L c'$.

Proof. By case analysis on event e. Case ϵ is trivial. Rule [Spawn] ensures that the child thread is at least as sensitive as the parent n_H, hence if $e = \mathbf{spawn}(\Delta, n_2, t, K)$, then n_2 is secret and furthermore $n_2 \notin P$, hence $c \approx_L c'$, because the global state c' changes only in parts that are not observable by the attacker, i.e., $T \approx_L T[n_2 \mapsto (\Delta, n_2.pc, \varnothing, [\,] \mid t, [\,])]$, $T \approx_L T[n_2 \mapsto s]$, $B \approx_L B[n_2 \mapsto B_0]$ and $H \approx_L H[n_2 \mapsto H_0]$, because erasure filters out the new secret binding $n_2 \notin P$. Furthermore, if the parent thread has observable resources, i.e., $n_H \in P$, then erasure reassigns the core capabilities of the child to the parent, i.e., $\theta[n_H \mapsto \theta(n_H) \setminus \{n_2.k\} \cup K][n_2 \mapsto K] \approx_L \theta$ (Figure 16b). Lastly, the core maps are L-equivalent, i.e., $\Phi \approx_L \Phi[n_2.k \mapsto \langle n_2^{B_0} \rangle]$, because $\Phi(n_2.k) \approx_L \langle n_2^{B_0} \rangle$ and $\varepsilon_L(\langle n_2^{B_0} \rangle) = \langle \circ^0 \rangle = \varepsilon_L(\Phi)(n_2.k)$, since $n_2 \notin P$ and $n_2.k$ is *free* in Φ (thread n_1 could have not spawned on that core otherwise). Case $\mathbf{fork}(\Delta, n_2, t, b, h)$ is similar. Case $\mathbf{kill}(n')$ is trivial, if $n' \notin Dom(T)$. Otherwise, we observe that $n' \notin P$ and so are its descendants, i.e., $i \notin P$ where $i \in N$ and $N = [\![\{n'\}]\!]^T$, and the changes to the budget maps involve only *secret* threads. If the parent thread has observable resources, i.e., $n_H \in P$, then the new budget maps are also indistinguishable to the attacker, e.g., $B[n \mapsto B(n) + \sum_{i \in N, i.k = n.k} B(i)] \approx_L B$. The final core map is L-equivalent to the initial map, i.e., $\Phi(k) \approx_L \Phi(k) \setminus N$ for $k \in \{1 \,..\, \kappa\}$, because N contains only *secret* threads and their left-over budget is reassigned to their closest ancestor in P. Lastly, we prove case $\mathbf{send}(n_2, t)$ by case analysis on the guards of function $next(\cdot)$. The lemma follows trivially in the first and in the third case—the global state does not change because the message is dropped. In the second case, the message gets delivered in the message queue of thread n_2, under the condition that $pc_1 \sqsubseteq pc_2$, i.e., $pc_2 \equiv H$. If $n_2 \notin P$, erasure drops its configuration from the thread pool and we obtain L-equivalence. If $n_2 \in P_H$, then $\varepsilon_L(T)(n_2) = (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$ and the update does not change the thread pool, because $\bullet \triangleright t \equiv \bullet$ and $\bullet \cup \Delta \equiv \bullet$.

Lemma 8. *If* $next(n_L, e, c) = c'$, *then* $next(n_L, \varepsilon_L(e), \varepsilon_L(c)) = \varepsilon_L(c')$.

Proof. Intuitively, the lemma ensures that processing any event e generated by *public* thread $n_H \in P$ commutes with the erasure function. The proof starts with case analysis on e. Case ϵ is trivial. Two cases follow for event $\mathbf{spawn}(\Delta, n_2, t, K)$ depending on the security level of the child thread. If the thread is *public*, i.e., $T(n_2).pc \equiv L$, then erasure rewrites the event to $\mathbf{spawn}(\varepsilon_L(\Delta), n_2, \varepsilon_L(t), K)$ and the updates to the budget and core maps commute with erasure, e.g., $\varepsilon_L(T[n_2 \mapsto (\Delta, L, \varnothing, [\,] \mid t, [\,])]) = \varepsilon_L(T)[n_2 \mapsto (\varepsilon_L(\Delta), L, \varnothing, [\,] \mid \varepsilon_L(t), [\,])]$. If the child thread is *secret*, i.e., $T(n_2).pc \equiv H$, we apply *2-steps erasure* to simulate the sensitive write operation [153]. In particular, erasure rewrites the event to special event $\mathbf{spawn}_L(\varepsilon_L(\Delta), n_2, K)$, and function $next(\cdot)$ handles it so that it gives global state $\varepsilon_L(c')$ (Fig. 16c). Notice that the child thread has a secret current label and its parent is public, thus $n_2 \in P_H$ and $\varepsilon_L(T[n_2 \mapsto (\Delta, L, \varnothing, [\,] \mid t, [\,])]) = \varepsilon_L(T)[n_2 \mapsto (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)]$. Case $\mathbf{fork}(\Delta, n_2, t, b, h)$ is similar. In case $\mathbf{kill}(n_2)$, we apply Lemma 6 to show that the resources assigned to the parent thread n_L remain the same under erasure. In case $\mathbf{send}(n_2, t)$, we distinguish two cases depending on the sensitivity of n_2. If *public*, i.e., $T(n_2).pc \equiv L$, then the lemma is trivial. In function $next(\cdot)$ the same guard holds true under erasure, i.e., $n_L \in Dom(T)$ iff $n_L \in Dom(\varepsilon_L(T))$, $\varepsilon_L(T)(n_L) = \varepsilon_L(T(n_L))$ and $|\Delta| = |\varepsilon_L(\Delta)|$ (Lemma 5.1), and the public updates to the thread pool commutes under erasure. If the recipient is *secret*, i.e., $T(n_2).pc \equiv H$, then we further distinguish between $n_2 \in P_H$, which follows as before, and $n_2 \notin P_H$, which implies that $n_2 \notin Dom(\varepsilon_L(T))$, the message is dropped and the thread map remains unchanged, i.e., $\varepsilon_L(T) \equiv \varepsilon_L(T[n_2 \mapsto s])$.

By means of this lemma, we show that erasure commutes with processing parallel events *commute*.

Lemma 9. $\varepsilon_L(\langle\!\langle sort\ es \rangle\!\rangle^c) \equiv \langle\!\langle sort\ (map\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)}$

Proof. Intuitively, the proof relies on the fact that *public* events, i.e., events generated by public threads, are erased homomorphically, i.e., $\varepsilon_L(n_L, e) = (n_L, \varepsilon_L(e))$ and *secret* events are rewritten to ϵ, i.e., $\varepsilon_L(n_H, e) = (\_, \epsilon)$ otherwise.[23] The proof follows by induction on the event list es. The base case is trivial. In the inductive case, we need to consider how event erasure affects sorting. Intuitively, the erasure function does not affect the relative order of public events, i.e., (n_L, e), because $\varepsilon_L(e)$ has the same priority as e. Instead, secret events end up at the end of the list because their event is erased to ϵ, which has the least priority. If the next event is *secret*, we use Lemma 7 and we apply induction, after removing the corresponding event in the erased list—the erased event is trivial (ϵ) and it does not change the state, i.e., $\langle\!\langle sort\ ((n_H, \epsilon) : map\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)} \equiv \langle\!\langle sort\ (map\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)}$. If the next event is *public*, then the scheduler processes the corresponding erased

[23] The thread that generates ϵ could be different from n_H, see Proposition 7.2 and 7.3. Since the event ϵ has no effects on the state, the thread identifier is irrelevant for the rest of the proof.

event in the erased configuration. The proof then follows by Lemma 8 and straightforward induction.

We conclude this section by showing that hierarchical scheduling (Figure 17a) is deterministic.

Lemma 10. *If* $Q[\langle n_1^{b_1}\rangle]$ *and* $Q[\langle n_2^{b_2}\rangle]$, *then* $n_1 \equiv n_2$ *and* $b_1 \equiv b_2$.

Proof. Induction on the scheduling relation.

B.3 Progress-Insensitive Non-interference

Using the auxiliary lemmas listed above, we prove *progress-inensitive* noninterference, which guarantees that L-equivalence is preserved under assumptions that both configurations make progress. As explained in Section 5.2, the property relies on*determinism* of the stepping relation and *simulation*.

Proposition 5 (Determinism). *For all states* s_1, s_2, s_3, *core queues* Q_1, Q_2, Q_3, *thread ids* n_1, n_2, *events* e_1, e_2, *global states* Σ, *configurations* c_1, c_2, c_3, *the following hold:*

1. *If* $s_1 \rightsquigarrow_\mu s_2$ *and* $s_1 \rightsquigarrow_\mu s_3$, *then* $s_2 \equiv s_3$.
2. *If* $Q_1 \xrightarrow{(n_1,s_1,e_1)}_\Sigma Q_2$ *and* $Q_1 \xrightarrow{(n_2,s_2,e_2)}_\Sigma Q_3$, *then* $n_1 \equiv n_2$, $s_1 \equiv s_2$, $e_1 \equiv e_2$ *and* $Q_2 \equiv Q_3$.
3. *If* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$, *then* $c_2 \equiv c_3$.

Proof.

1. Case analysis on the sequential step relation.
2. By Lemma 10, we derive $n_1 \equiv n_2$, thus the same thread is scheduled on both cores. Then, we do case analysis and, since Σ is the same in both reductions, the threads step likewise, i.e., $s_1 \equiv s_2$ (using Lemma 5.1 in case [Step]), and generate the same event, $e_1 \equiv e_2$. The resulting cores are equal, i.e., $Q_1 \equiv Q_2$, since both threads have the same residual budget (Lemma 10).
3. The lemma follows immediately by applying determinism, i.e., Lemma 5.2, on all the core reductions $\Phi(i) \xrightarrow{(n_i,s_i,e_i)}_\Sigma Q_i$, for $i \in \{1\,..\,\kappa\}$.

Proposition 6 (Sequential Simulation). *If* $s \rightsquigarrow s'$, *then* $\varepsilon_L(s) \rightsquigarrow \varepsilon_L(s')$.

Proof. If the current label is H, then simulation follows by rule [Hole], which simply ticks, i.e., $(\bullet,\bullet,\bullet,\bullet \mid \bullet,\bullet) \rightsquigarrow (\bullet,\bullet,\bullet,\bullet \mid \bullet,\bullet)$. If the current label is L, then simulation follows by applying the same step rule under reduction. For those, we discuss only the interesting cases, where we apply 2-steps erasure [153]—all the others cases are trivial and follow by applying Lemma 5 when needed, e.g., 5.1 for rule [App$_2$] and 5.3 for rule [GC]. The erasure function rewrites term *label* to special term $label_L$, then rule [Label$_{L1}$] simulates [Label$_1$] and rule [Label$_{L2}$] ([Label$_{L3}$]) simulates [Label$_2$] when the assigned label is *public*, i.e., L, (resp. *secret*, i.e., H). Similarly, erasure

rewrites term $unlabel$ to special term $unlabel_L$, then rule [UNLABEL$_{L1}$] simulates [UNLABEL$_1$] and rule [UNLABEL$_{L2}$] ([UNLABEL$_{L3}$]) simulates [UNLABEL$_2$], when the labeled value is *public*, i.e., $Labeled\ L\ \varepsilon_L(t)^\circ$ for some term t (resp. *secret*, i.e., $Labeled\ H\ \bullet$).

We now lift *simulation* to core reductions. For brevity, we write thread's state $\bullet$ for erased state $(\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$, state $s_\circ$ for the spinning thread's state, i.e., $s_\circ = ([\,], \bot, \varnothing, [\,] \mid return\ (), [\,])$.

Proposition 7 (Core Simulation). *Given a core reduction step* $Q \xrightarrow{(n,s,e)}_\Sigma Q'$*, then one of the following holds:*

1. *If* $n \in P$*, then* $\varepsilon_L(Q) \xrightarrow{\varepsilon_L(n,s,e)}_{\varepsilon_L(\Sigma)} \varepsilon_L(Q')$
2. *If* $n \notin P$ *and* $Q \cap P = \varnothing$*, then* $\varepsilon_L(Q) \xrightarrow{(\circ, s_\circ, \epsilon)}_{\varepsilon_L(\Sigma)} \varepsilon_L(Q')$
3. *If* $n \notin P$ *and* $Q \cap P \neq \varnothing$*, then there exists some thread* $n' \in P_H$*, such that* $\varepsilon_L(Q) \xrightarrow{(n', \bullet, \epsilon)}_{\varepsilon_L(\Sigma)} \varepsilon_L(Q')$

Proof.

1. The first case simulates the execution of a thread with visible resources, i.e., $n \in P$, which executes similarly under erasure. We distinguish two cases depending on the thread's current label.
   - If $n_L \in P_L$, we show that the same thread is scheduled using Lemma 2 and we proceed by case analysis on the core step. Since the thread's current label is public, erasure preserves the thread's structure and its resources. As a result the erased thread steps likewise, i.e., it performs exactly the same reduction step. In particular, we apply Proposition 6 in case [STEP] and Lemma 1 in case [CONTEXTSWITCH].
   - If $n_H \in P_H$, we apply Lemma 3, i.e., the core scheduler executes the same secret thread, which then *ticks*, i.e., it reduces with rule [STEP] applied to rule [HOLE], since its configuration gets completely erased, i.e., $\varepsilon_L(T(n_H)) = (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$.
2. The second case simulates the execution of a core allocated to threads with no visible resources, i.e., $n \notin P$ and $Q \cap P = \varnothing$. This case occurs if some secret thread with public resources *spawns* another secret thread on a core that it owns. Under erasure, the core is still *free*, i.e., $\varepsilon_L(Q) = \langle \circ^0 \rangle$ and the spinning thread $\circ$ takes over in the erased core queue via rule [CONTEXTSWITCH].
3. The third case simulates the execution of a secret thread with no visible resources, i.e., $n \notin P$, which shares the core with threads gets scheduled on the queue Q, which contains some other thread with visible resources, i.e., $Q \cap P \neq \varnothing$. For example, this happens when a secret thread with visible resources *forks* another secret thread on the same core. Then, we apply Lemma 4 and conclude that the core scheduler executes its closest

ancestor $n'_H \in P_H$, which remains in the erased core. [24] Thread n'_H simulates the execution of thread n_H by rule [Step] and [Hole] (see above). We remark that core erasure cancels out the effects of fork from queue Q' (rule [Fork]), as it collapses both thread n_H and its new child to the ancestor thread n'_H, which regains their resources.

Proposition 2 (Parallel Simulation). If $c \hookrightarrow c'$, then $\varepsilon_L(c) \hookrightarrow \varepsilon_L(c')$.

Proof. The proof requires to show that running the parallel scheduler on the erased initial state, i.e., $\varepsilon_L(c)$, by means of rule [Parallel] from Fig. 5, gives a final state that corresponds to the erased state obtained in original step i.e., $\varepsilon_L(c')$. The fact that the core scheduler could schedule different threads on erased cores makes the proof interesting. Intuitively, public cores, i.e., cores that execute public threads, proceed in lock-step with the original configuration (Proposition 7.1) and the parallel scheduler processes their public events in the same relative order, so that erasure and changes to the state *commute*. Instead, secret cores execute either the *public* spinning thread (Proposition 7.2) or a another public thread $n'_H \in P_H$ (Proposition 7.3). In either case, they generate the trivial event, i.e., ϵ, which leaves the global state unchanged, hence the different order with which they get processed is irrelevant under erasure.

Firstly, we apply *core simulation*, i.e., Proposition 7, to the core scheduler step on each core $i \in \{1 \,..\, \kappa\}$. Then, the parallel scheduler changes the global state accordingly: it updates the erased thread pool $\varepsilon_L(T)$, the core map $\varepsilon_L(\Phi)$ and then processes the events generated by the core scheduler. Those operations either *commute* under erasure, whenever they affect *public* parts of the state, or have *no effect*, otherwise. We show that, by case analysis on the security level of the threads scheduled in the erased configuration. In particular, for each thread n_i that runs in the original configuration, one of the clauses of Proposition 7 applies.

Commutativity holds in the first case (Proposition 7.1), since $\varepsilon_L(T[n \mapsto s]) \equiv \varepsilon_L(T)[n \mapsto \varepsilon_L(s)]$, when $n \in P$. In the second case (Proposition 7.2), we show $\varepsilon_L(T[n \mapsto s]) \equiv \varepsilon_L(T)[\circ \mapsto \varepsilon_L(s_\circ)]$ by rewriting both sides of the equation to $\varepsilon_L(T)$. On the left-hand side, erasure filters out the secret thread from T, since $n \notin P$, while the update does not change the thread pool on the right-hand side, because $\varepsilon_L(T)(\circ) \equiv \varepsilon_L(s_\circ) \equiv ([\,], \bot, \varnothing, [\,] \mid \mathit{return}\ (), [\,])$ In the third case (Proposition 7.3), the ancestor $n' \in P$ takes over n in the erased core and the update to the thread map has no effect, i.e., $\varepsilon_L(T) = \varepsilon_L(T[n' \mapsto \bullet])$, since $\varepsilon_L(T)(n'_H) = (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$. In all cases, core map erasure is homomorphic, i.e., $\varepsilon_L(\Phi) = \lambda k.\varepsilon_L(\Phi(k))$, then core map updates always commute with erasure, i.e., $\varepsilon_L(\Phi[k \mapsto Q_i]) \equiv \varepsilon_L(\Phi)[k \mapsto \varepsilon_L(Q_i)]$. To conclude the proof, we apply Lemma 9 and show that erasing and processing events *commute*, i.e., $\varepsilon_L(\langle\!\langle\, \mathit{sort}\ \mathit{es} \,\rangle\!\rangle^c) \equiv \langle\!\langle\, \mathit{sort}\ (\mathit{map}\ \varepsilon_L(\cdot)\ \mathit{es}) \,\rangle\!\rangle^{\varepsilon_L(c)}$.

We conclude with the proof of *progress-insensitive* noninterference.

[24] The current label of the ancestor cannot be public, i.e., $n'_L \in P_L$), otherwise $n_H \in P_H$ and case 1 applies instead.

Proposition 3 (Progress-Insensitive Non-interference). *If* $c_1 \hookrightarrow c_1'$, $c_2 \hookrightarrow c_2'$ *and* $c_1 \approx_L c_2$, *then* $c_1' \approx_L c_2'$.

Proof. We apply *parallel simulation*, i.e., Proposition 2 and derive the erased reductions $\varepsilon_L(c_1) \hookrightarrow \varepsilon_L(c_1')$ and $\varepsilon_L(c_2) \hookrightarrow \varepsilon_L(c_2')$. From $c_1 \approx_L c_2$, we obtain $\varepsilon_L(c_1) \equiv \varepsilon_L(c_2)$ (Definition 1), and Proposition 5.3 (*parallel determinism*) gives $\varepsilon_L(c_1') \equiv \varepsilon_L(c_2')$, i.e., $c_1' \approx_L c_2'$.

B.4 Timing-Sensitive Non-interference

We now lift the security guarantees of $\mathbf{LIO}_{\mathrm{PAR}}$ to be *timing-sensitive*. Intuitively, timing-insensitive non-interference ensures that the timing of any parallel program does not depend on secret information, when executed with $\mathbf{LIO}_{\mathrm{PAR}}$ runtime system. More precisely, the theorem ensures that if a parallel program steps with some secret information, then it also steps with different secret information. The key property to proving this stronger form of non-interference is *time-sensitive* progress, i.e., Proposition 4, which reconstructs the *single step* taken by program in the other execution. This property relies on some side conditions that guarantee that the program satisfies some basic correctness properties, that we formally specify in the definition of *valid configuration.*

Definition 2 (Valid Configuration). *A configuration* $\langle T, B, H, \theta, \Phi, \omega \rangle$ *is valid if and only if it satisfies the following properties:*

- *If* $T(n) = s$, *then state* s *is well-formed, i.e., it is the state of a well-typed thread and* $x \in Dom(s.\Delta)$ *for all variables* $x \in fv(s)$;
- *For all cores* $k \in \{1 \mathinner{..} \kappa\}$, $\circ_k \mapsto s_\circ \in T$, $\circ_k \mapsto 0 \in B$, $\circ_k \mapsto H_0 \in H$, $\circ_k \mapsto \varnothing \in \theta$;
- $Dom(T) = Dom(B) = Dom(H) = Dom(\theta)$;
- *If* $n \in Q$, *then* $n \in Dom(T)$;
- *If* $Q[\langle n^b \rangle]$, *then* $b \leqslant B(n)$, $|T(n).\Delta| \leqslant H(n)$ *and* $n.k \notin \theta(n)$;
- *For all threads* $n_1\ n_2 \in Dom(T)$, *such that* $n_1 \neq n_2$, $\theta(n_1) \cap \theta(n_2) = \varnothing$;

It is easy to see that the initial parallel configuration (Corollary 1) is *valid*. The next lemma shows that valid configurations that execute with the operational semantics of $\mathbf{LIO}_{\mathrm{PAR}}$ remain valid.

Lemma 11 (Valid Invariant).
For all valid configurations c, *if* $c \hookrightarrow c'$, *then* c' *is valid.*

Proof. By case analysis on all the reduction relations.

Proposition 4 (Time-Sensitive Progress). *For all valid configurations* c_1, c_1' *and* c_2 *and parallel reduction steps* $c_1 \hookrightarrow c_1'$, *if* $c_1 \approx_L c_2$, *then there exists a configuration* c_2', *such that* $c_2 \hookrightarrow c_2'$.

Proof. The parallel scheduler has only one reduction rule, i.e., [Parallel], thus

the proof mainly relies on *core progress*, i.e., showing that for each core that steps in the first configuration, then the corresponding core in the second configuration also steps. Formally, for all *valid* cores $Q_1 \approx_L Q_2$ and global states $\Sigma_1 \approx_L \Sigma_2$, if $Q_1 \xrightarrow{m_1}_{\Sigma_1} Q_1'$ and $Q_1 \approx_L Q_2$, then there exists Q_2' and m_2, such that $Q_2 \xrightarrow{m_2}_{\Sigma_2} Q_2'$. Since $c_1 \approx_L c_2$, the core maps in the configurations are L-equivalent, i.e., $\Phi_1 \approx_L \Phi_2$, hence $\Phi_1(i) \approx_L \Phi_2(i)$, for all $i \in \{1 \,..\, \kappa\}$. In particular, we apply *core simulation*, i.e., Proposition 7, which gives us $\varepsilon_L(Q_1) \xrightarrow{m_3}_{\varepsilon_L(\Sigma_1)} \varepsilon_L(Q_1')$, for some message m_3, and use that together with $\varepsilon_L(Q_1) \equiv \varepsilon_L(Q_2)$ and the assumption that Q_1 and Q_2 are valid to reconstruct Q_2', m_2 and the other step $Q_2 \xrightarrow{m_2}_{\varepsilon_L(\Sigma_2)} Q_2'$.

The task of reconstructing a core step is facilitated by the fact that the core semantics always steps, regardless of the number of threads on the core, their resources and state. Specifically, either threads execute *sequentially* ([Step]), or they perform a *concurrent* operation ([Fork,Spawn,Wait]), or they are *stuck* [Stuck] otherwise. Note that, even if no thread has sufficient time budget to step, then rule [ContextSwitch] takes over and that even *free* cores step thanks to the core's *spinning* thread, i.e., $\circ$.

Theorem 1 (Timing-Sensitive Non-interference). *For all valid configurations c_1 and c_2, if $c_1 \hookrightarrow c_1'$ and $c_1 \approx_L c_2$, then there exists a configuration c_2', such that $c_2 \hookrightarrow c_2'$ and $c_1' \approx_L c_2'$*

Proof. We apply time-sensitive progress, i.e., Proposition 4 to valid configurations $c_1 \approx_L c_2$ and obtain the second reduction $c_2 \hookrightarrow c_2'$ for some configuration c_2'. We then derive L-equivalence of the final configurations, i.e., $c_1' \approx_L c_2'$, from progress-insensitive non-interference, i.e., Proposition 3, applied to $c_1 \hookrightarrow c_1'$, $c_2 \hookrightarrow c_2'$ and $c_1 \approx_L c_2$.

Corollary 1 generalizes timing-sensitive non-interference to many steps by applying Theorem 1 and Proposition 11 as many times.

C Attack Code

Below we list the code for the parallel scheduler-based attacks. The code depends on the following external packages:

```
- lio-0.11.6.0
- hashable-1.2.7.0
- text-1.2.3.1
- array-0.5.1.1
- bytestring-0.10.8.1
```

Haskell package manager Cabal can be used to configure and install these dependencies with the command `cabal install packages`. Both attack modules rely on some helper functions found in Appendix C.3. Additionally, both attacks rely on thresholds that must be determined empirically as they are machine dependent.

The attacks can be compiled and executed using GHC version 8.0.2 with the following commands, where `CODE` is the file containing the attack code and `SECRET` represents the secret value to leak (*0* or *1*),

```
$ ghc -threaded -rtsopts CODE lib.hs -o attack
$ ./attack SECRET +RTS -N2 -RTS
```

Section C.1 and C.2 list the code of the reclamation and allocation attacks (`reclamation.hs` and `allocation.hs`), sketched in Section 2.3. In the attacks, a secret thread affects the CPU-time available to other public threads by terminating early (the scheduler *reclaims* its quota of CPU-time), or forking another secret thread (the scheduler *allocates* a new CPU-time quota). The attacks are written using the **LIO** library and disjunction category (`DC`) labels to specify the security lattice [142].

C.1 Reclamation Attack

```
-- reclamation.hs

module Main where

import System.Environment
import Data.List

import LIO
import LIO.LIORef
import LIO.DCLabel
import LIO.Concurrent
import LIO.Run

import Lib

-- Thresholds.
-- These parameters are machine specific and estimated empirically.
len        = 100000
threshold  = 3000

-- The code run by the secret thread.
-- If secret == 1, the thread loops,
-- otherwise it terminates right away.
highThread :: DC (DCLabeled Int) -> DC Int
highThread secret = do
  s <- unlabel secret
  case s of
    1 -> do
      t1 <- busyWait 100000
      return 0
    _ -> return 0

-- Simple heuristic to determine the value of the secret
analyze :: (String, Int) -> Int
analyze (_,a) = if a > threshold then 1 else 0

-- Count the number of messages from each thread
-- in the public channel and infer the secret.
count :: LIORef DCLabel [String] -> DC Int
count channel= do
  msgs <- readLIORef channel
  let acc = map (\x -> (head x, length x)) (group msgs)
  return $ analyze $ head acc

-- Fork the two public threads that write to
-- the same public channel
```

```
runLowThreads :: LIORef DCLabel [String] -> DC Int
runLowThreads channel = do
  t1 <- lFork low (writeA len channel)
  t2 <- lFork low (writeB len channel)
  () <- lWait t1
  () <- lWait t2
  -- analyze the public channel to infer the secret
  count channel

-- Start the the secret thread and the two public threads,
-- which return the secret.
leak :: DC (DCLabeled Int) -> DC Int
leak secret = do
  channel <- newLIORef low []
  secretThread <- lFork secretL (highThread secret)
  secretValue <- runLowThreads channel
  return secretValue

main :: IO ()
main = do
  l <- getArgs
  let secret = getArg l
  secret <- evalLIO (leak (label secretL secret)) init
  print $ "The secret is " ++ (show secret)
  return ()
```

C.2 Allocation Attack

```
-- allocation.hs

module Main where

import System.Environment
import Data.List

import LIO
import LIO.LIORef
import LIO.DCLabel
import LIO.Concurrent
import LIO.Run

import Lib

-- Thresholds.
-- These parameters are machine specific and estimated empirically.
len           = 100000
lowThreshold  = 4000
highThreshold = 15000

-- The code run by the secret thread.
-- If secret == 1, then the thread forks a child,
-- otherwise it loops.
highThread :: DC (DCLabeled Int) -> DC Int
highThread secret = do
  s <- unlabel secret
  case s of
    1 -> do
      s1 <- lFork high (busyWait len)
      t1 <- busyWait len
      res <- lWait s1
      return 0
    _ -> do
      t1 <- busyWait len
      return 0

-- Simple heuristic to determine the value of the secret.
analyze :: (String, Int) -> Int
analyze (_,a) =
  if (a < lowThreshold) || (a > highThreshold)
    then 1
    else 0

-- Count the number of messages from each thread
-- in the public channel and infer the secret.
```

```
count :: LIORef DCLabel [String] -> DC Int
count channel= do
  msgs <- readLIORef channel
  let acc = map (\x -> (head x, length x)) (group msgs)
  return $ analyze $ head acc

-- Fork the two public threads that write
-- to the same public channel
runLowThreads :: LIORef DCLabel [String] -> DC Int
runLowThreads channel = do
  t1 <- lFork low (writeA len channel)
  t2 <- lFork low (writeB len channel)
  () <- lWait t1
  () <- lWait t2
  -- analyze the public channel to infer the secret
  count channel

leak :: DC (DCLabeled Int) -> DC Int
leak secret = do
  channel <- newLIORef low []
  secretThread <- lFork high (highThread secret)
  secretValue <- runLowThreads channel
  return secretValue

main :: IO ()
main = do
  l <- getArgs
  let secret = getArg l
  secret <- evalLIO (leak (label high secret)) init
  print $ "The secret is " ++ (show secret)
  return ()
```

C.3 Helper Functions

```
-- lib.hs

module Lib where

import LIO
import LIO.LIORef
import LIO.DCLabel

import Control.Monad

-- Command line parsing
getArg []    = -1
getArg (a:r) = read a

-- Labels
low = "Public" %% "Public"
secretL = "Secret" %% "Secret"
high = low `lub` secretL

-- Initial LIO state (current label and clearance)
init = LIOState { lioLabel     = low
                , lioClearance = high }

-- Write a message to the channel for a given number of times.
write :: String -> Int -> LIORef DCLabel [String] -> DC ()
write msg n ref = replicateM_ n trace
  where trace = do
          () <- modifyLIORef ref (\l -> msg:l)
          return ()

-- Write "A" to the channel
writeA :: Int -> LIORef DCLabel [String] -> DC ()
writeA len ref = write "A" len ref

-- Write ``B" to the channel
writeB :: Int -> LIORef DCLabel [String] -> DC ()
writeB len ref = write "B" len ref

-- Busy waiting.
busyWait :: Int -> DC Int
busyWait 0 = return 1
busyWait n = do
  acc <- busyWait (n - 1)
  return $ acc + n
```

Bibliography

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM.
2. Maximilian Algehed and Alejandro Russo. Encoding dcc in haskell. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 77–89, New York, NY, USA, 2017. ACM.
3. A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 137–151, July 2015.
4. Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.
5. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.
6. Joshua Auerbach, David F Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 245–254. ACM, 2008.
7. Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.
8. Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.
9. Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.
10. Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.
11. Jean Bacon, David M. Eyers, Thomas F. J.-M. Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter R. Pietzuch. Information flow control for secure cloud computing. *IEEE Transactions on Network and Service Management*, 11:76–89, 2014.
12. Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 162–173, New York, NY, USA, 2000. ACM.
13. Henry G Baker Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

14. Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld. Jslinq: Building secure applications across tiers. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 307–318, New York, NY, USA, 2016. ACM.
15. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, March 2005.
16. Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Computer Languages, Systems & Structures*, 33(2):35–59, 2007.
17. Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *Special issue of ACM Transactions on Information and System Security (TISSEC)*, 2009.
18. Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Annual Network & Distributed System Security Symposium*. Internet Society, 2015.
19. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
20. K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.
21. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 159–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
22. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 159–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
23. Guy E Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *ACM SIGPLAN Notices*, volume 34, pages 104–117. ACM, 1999.
24. Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*. ACM, 2007.
25. William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 101–113, 2015.
26. Joachim Breitner. dup – explicit un-sharing in haskell. *CoRR*, abs/1207.2017, 2012.
27. Joachim Breitner. Formally proving a compiler transformation safe. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 35–46, New York, NY, USA, 2015. ACM.
28. Niklas Broberg, Bart Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 217–232, Berlin, Heidelberg, 2013. Springer-Verlag.
29. Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Conference on Security*, SEC. USENIX Association, 2013.
30. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.

31. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proceedings of the 18th Nordic Conference on Secure IT Systems - Volume 8208*, NordSec 2013, pages 116–122, New York, NY, USA, 2013. Springer-Verlag New York, Inc.
32. Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.
33. Winnie Cheng, Dan R.K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 139–151, Boston, MA, 2012. USENIX.
34. Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 31–44, October 2007. (Best paper award.).
35. Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.
36. Ankush Das and Jan Hoffmann. ML for ML: Learning cost semantics by experiment. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 190–207, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
37. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communication of the ACM*, 20(7):504–513, July 1977.
38. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.
39. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *ACM Symposium on Operating Systems Principles*, SOSP. ACM, 2005.
40. Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1-3):35–75, December 1991.
41. Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.
42. L. Fennell and P. Thiemann. Gradual security typing with references. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 224–239, June 2013.
43. Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium*, pages 531–548, 2016.
44. Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27, 2016.
45. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

46. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. *Journal of Computer Security*, 25, 2017.
47. J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.
48. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy*, pages 75–75, April 1984.
49. Helena Handschuh and Howard M. Heys. A timing attack on RC5. In *Proceedings of the Selected Areas in Cryptography*, SAC '98, pages 306–318, Berlin, Heidelberg, 1999. Springer-Verlag.
50. D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 351–365, July 2015.
51. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing*. ACM, 2014.
52. Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, pages 319–347, 2012.
53. Daniel Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.
54. Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, 1998.
55. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*, pages 11–31, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
56. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
57. Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure Information Flow as Typed Process Behaviour. In *European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.
58. Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 81–92, New York, NY, USA, 2002. ACM.
59. C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
60. Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C. Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 398–410, Washington, DC, USA, 2014. IEEE Computer Society.
61. Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4):233–254, 1992.

62. J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.
63. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, pages 170–178, Berlin, Heidelberg, 2012. Springer-Verlag.
64. Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android. In *Computer Security – ESORICS 2013*, pages 775–792, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
65. Richard A Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems (TOCS)*, 1(3):256–277, 1983.
66. Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4):291–347, December 2005.
67. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman,RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.
68. David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, pages 463–480, 2016.
69. Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP. ACM, 2007.
70. Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.
71. Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. *ACM SIGPLAN Notices*, 49(6):2–14, 2014.
72. B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.
73. John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM.
74. Jaewoo Lee and Chris Clifton. How much is enough? choosing ϵ for differential privacy. In *Proceedings of the 14th International Conference on Information Security*, ISC'11, pages 325–340, Berlin, Heidelberg, 2011. Springer-Verlag.
75. Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 16–, Washington, DC, USA, 2006. IEEE Computer Society.
76. Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, April 2010.
77. Ximeng Li, Heiko Mantel, and Markus Tasch. Taming message-passing communication in compositional reasoning about confidentiality. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, pages 45–66, 2017.
78. S. Lipner, T. Jaeger, and M. E. Zurko. Lessons from vax/svs for high-assurance vm systems. *IEEE Security Privacy*, 10(6):26–35, Nov 2012.

79. Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.
80. Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.
81. Jed Liu and Andrew C Myers. Defining and enforcing referential security. In *International Conference on Principles of Security and Trust*, pages 199–219. Springer, 2014.
82. Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328, New York, NY, USA, 2015. ACM.
83. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 218–232, June 2011.
84. Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Comput. Secur.*, 11(4):615–676, July 2003.
85. Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European Conference on Research in Computer Security*, ESORICS'10, pages 116–133, Berlin, Heidelberg, 2010. Springer-Verlag.
86. Simon Marlow. *Parallel and concurrent programming in Haskell*. O'Reilly, July 2013.
87. Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *ACM SIGPLAN Notices*, 46(12):71–82, 2012.
88. Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.
89. Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.
90. D. McCullough. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy(SP)*, volume 00, page 161, April 1987.
91. Simon Meurer and Roland Wismüller. Apefs: An infrastructure for permission-based filtering of android apps. In AndreasU. Schmidt, Giovanni Russello, Ioannis Krontiris, and Shiguo Lian, editors, *Security and Privacy in Mobile Information and Communication Systems*, volume 107. Springer Berlin Heidelberg, 2012.
92. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
93. S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 146–160, June 2011.
94. Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.
95. Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, pages 57–74, New York, NY, USA, October 2012. ACM Press.

96. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. *2012 IEEE Symposium on Security and Privacy*, 0, 2013.
97. Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE Computer Security Foundations Symposium*, pages 417–431, Lisbon, Portugal, June 2016.
98. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming*, pages 269–281, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.
99. Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.
100. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.
101. Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *USENIX Security Symposium*, pages 1119–1136, 2016.
102. Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, SP. IEEE Computer Society, 2011.
103. Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.
104. Stephen C North and John H Reppy. Concurrent garbage collection on stock hardware. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–133. Springer, 1987.
105. James Parker, Niki Vazou, and Michael Hicks. LWeb: Information flow security for multi-tier web applications. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2019.
106. James Lee Parker. *LMonad: Information flow control for haskell web applications*. PhD thesis, University of Maryland, College Park, 2014.
107. Mathias V. Pedersen and Aslan Askarov. From trash to treasure: Timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 693–709, 2017.
108. Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.
109. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
110. Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.
111. Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):1–255, Jan 2003. `http://www.haskell.org/definition/`.
112. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

113. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. http://www.cis.upenn.edu/ bcpierce/sf.
114. Filip Pizlo, Antony L. Hosking, and Jan Vitek. Hierarchical real-time garbage collection. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 123–133, New York, NY, USA, 2007. ACM.
115. François Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.
116. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.
117. Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. Progress-sensitive security for spark. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, pages 20–37, Berlin, Heidelberg, 2016. Springer-Verlag.
118. Willard Rafnsson, Limin Jia, and Lujo Bauer. Timing-sensitive noninterference through composition. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 3–25, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
119. Willard Rafnsson, Keiko Nakata, and Andrei Sabelfeld. Securing class initialization in Java-like languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1), January 2013.
120. Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. Type systems for information flow control: The question of granularity. *ACM SIGLOG News*, 4(1):6–21, February 2017.
121. Vineet Rajani and Deepak Garg. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *Proc. of the IEEE Computer Security Foundations Symp.*, CSF '18. IEEE Computer Society, 2018.
122. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. ACM, 2009.
123. Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 280–288, New York, NY, USA, 2015. ACM.
124. Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 13–24, New York, NY, USA, 2008. ACM.
125. Alejandro Russo and Andrei Sabelfeld. Security for multithreaded programs under cooperative scheduling. In Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, pages 474–480, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
126. Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *The Journal of Logic and Algebraic Programming*, 78(7):593 – 618, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007).
127. Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations*

Symposium, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

128. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.
129. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, CSFW '00, pages 200–, Washington, DC, USA, 2000. IEEE Computer Society.
130. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, Mar 2001.
131. Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1617–1634, New York, NY, USA, 2018. ACM.
132. Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca Viganò, editors, *POST*, volume 9635 of *LNCS*. Springer, 2016.
133. Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. Selinq: Tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 25–38, New York, NY, USA, 2014. ACM.
134. David Schultz and Barbara Liskov. Ifdb: Decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 43–56, New York, NY, USA, 2013. ACM.
135. Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, May 1997.
136. Naokata Shikuma and Atsushi Igarashi. Proving noninterference by a fully complete translation to the simply typed λ-calculus. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06, pages 301–315, Berlin, Heidelberg, 2007. Springer-Verlag.
137. V. Simonet. The Flow Caml system. Software release at http://cristal.inria.fr/ simonet/soft/flowcaml/, 2003.
138. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.
139. Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using JOANA. *it - Information Technology*, 56(6):280–287, 2014.
140. Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 718–735, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
141. Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent

information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012. ACM.

142. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Nordic Conference on Security IT Systems (NordSec)*. Springer, October 2011.
143. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457, to appear in Journal of Functional Programming*, 2012.
144. Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27, 2017.
145. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.
146. Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014.
147. David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. *SIGPLAN Not.*, 47(12):137–148, September 2012.
148. Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Transactions on Programming Languages and Systems*, 40(4):16:1–16:55, November 2018.
149. Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 187–202, Washington, DC, USA, 2007. IEEE Computer Society.
150. Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 115–125, New York, NY, USA, 2004. ACM.
151. Marco Vassena, Joachim Breitner, and Alejandro Russo. Securing concurrent lazy programs against information leakage. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 37–52, 2017.
152. Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 538–557, 2016.
153. Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 15–28, New York, NY, USA, 2016. ACM.
154. Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. Mac a verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming*, 2017.
155. Pepe Vila and Boris Kopf. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 849–864, Vancouver, BC, 2017. USENIX Association.

156. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*, pages 34–43, June 1998.
157. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.
158. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.
159. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.
160. Wing H. Wong. Timing attacks on rsa: Revealing your secrets through the fourth dimension. *XRDS*, 11(3):5–5, May 2005.
161. Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. Warding off timing attacks in deterland. In *Conference on Timely Results in Operating Systems, Monterey, CS, US*, 2015.
162. Edward Z. Yang and David Mazières. Dynamic space limits for haskell. *SIGPLAN Not.*, 49(6):588–598, June 2014.
163. Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *ACM SIGPLAN Notices*, volume 51, pages 631–647. ACM, 2016.
164. Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.
165. Stephan Arthur Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002. AAI3063751.
166. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.
167. Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.
168. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 563–574, New York, NY, USA, 2011. ACM.
169. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *ACM Conference on Programming Language Design and Implementation*. ACM, 2012.